8-2003

Techniques for publishing real-time data on a local area network

Victor Anderson

To the Graduate Council:

I am submitting herewith a thesis written by Victor Anderson entitled "Techniques for publishing real-time data on a local area network." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

B. Bomar, Major Professor

We have read this thesis and recommend its acceptance:

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Victor Anderson entitled "Techniques For Publishing Real-Time Data On A Local Area Network." I have examined the final paper copy of the thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. B. Bomar, Major Professor

We have read this thesis and
recommend its acceptance:

Dr. R. Joseph

Dr. M. Smith

Acceptance for the Council:

Vice Provost and Dean of Graduate Studies

TECHNIQUES FOR PUBLISHING REAL-TIME DATA ON A LOCAL AREA NETWORK

A Thesis

Presented for the

Master of Science

Degree

The University Of Tennessee, Knoxville

Victor Anderson

August 2003

DEDICATION

This thesis is dedicated to my wife Winifred Anderson for believing in me and encouraging me throughout my studies at the University of Tennessee Space Institute.

ACKNOWLEDGEMENT

I wish to thank those who helped me successfully complete my Master of Science degree in Electrical Engineering. I would like to thank Dr. Bomar for his guidance in exposing me to the concept of computer networks and systems. He tirelessly and patiently offered me invaluable insight throughout my research work. I would like to thank Dr. Joseph for supporting me in countless ways while I pursued my academic goals. He was instrumental in my getting to understand modern transform methods and random process theory. My gratitude also goes to Dr. Smith who introduced me to optical engineering and to Dr. Limbaugh for awakening my interest in numerical analysis.

My final word of thanks goes to my family and friends who have supported me in various ways.

ABSTRACT

The primary objective of this thesis was to develop techniques for publishing real-time data on a Local Area Network (LAN) using the two transport protocols, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). One of the main restrictions was that the Data Acquisition (DAQ) system, the server, which does time-critical functions, should not be interrupted as it sends the data to monitoring applications. It was also desired that the monitoring applications, the clients, receive most of the published data. The two protocols, TCP and UDP, were evaluated using programs developed in C/C++ and compiled with the Visual C 6.0 compiler under Windows 2000.

An ordinary TCP server software, developed in C/C++, published 100,000 1400 Byte TCP segments on a 100 Megabit per second (Mbps) LAN. An ordinary TCP client software was coded in C/C++ which read the published data. After reading 30,000 segments, the client application paused for 5 sec and the effect of the delay on the server was analyzed. The procedure failed to satisfy the requirement not to interrupt the server because the server stopped publishing data during the delay period. A proposed solution put the server's publishing procedure into a separate thread thereby isolating the server application as a whole from any client interference. The results proved this approach to be successful. It was also found that the server can maintain one-server-to-many-clients interaction if the bandwidth is reduced by the number of connected clients.

Using UDP sockets, ordinary server and client applications were developed to evaluate the real-time performance of the standard UDP sockets as was done for the TCP sockets. The client recorded the percentage of datagrams successfully received. Though the client did not block the server, it did not always receive 100% of the published datagrams. The server, however, could use UDP broadcast and publish simultaneously to multiple clients without reducing bandwidth. To solve the delivery reliability problem, the approach implemented allowed the UDP client to detect and request a retransmission of datagrams that it loses. The improved UDP server had the ability to retransmit datagrams that are lost by its clients. It was determined that the server efficiently serviced retransmission requests from the clients for data rates up to 9MB/s. In cases where the client application was engaged in other activities beside reading data

from the server, an ordinary UDP client read 100% of all published data only at rates below 4 MB/sec, while the improved client read 100% of data up to 6 MB/sec.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

One of the primary reasons for networking is the need for sharing data over the network among connected nodes or hosts. Applications on the host machines communicate with one another sharing data and information over the network or they communicate on the same host. Network applications use a form of communication known as the *client-server paradigm*. The *client* is the application that actively initiates a contact while the application that waits passively for the contact is called the *server*.

This thesis applies the client-server paradigm in solving a real-time problem. Given an embedded real-time data acquisition system that takes data and does time-critical processing, data logging, or other critical data processing, the processed data is also to be published on a Local Area Network (LAN) for periodic remote monitoring. We define the real-time system as the server, and the remote monitoring applications as clients. It is required that the server is never interrupted in executing its time-critical functions while it publishes the data. However, subject to this restriction it is also desired that as much of the published data as possible get to the client(s).

Like most application programs, the clients and server applications need to use a transport protocol to share the published data. The Transmission Control Protocol/Internet Protocol (TCP/IP) suite comes with two main transport protocols. They are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

Transmission Control Protocol is a connection-oriented stream protocol with mechanisms that try to ensure a reliable service over the IP layer, which is not reliable. The peers at both ends of a TCP connection exchange state information so they can detect if the data they sent do not get to intended destination, when to do a retransmission, and so on. Much research has gone into making TCP reliable, so that TCP guarantees safe data delivery from one peer to another, independent of their physical location on the network. With TCP, data can be safely sent around the world,

but for the purpose of publishing data on the LAN without interrupting the server in its time-critical functions, TCP can be a challenge due to the following:

1) Each connection cannot be safely terminated without mutual agreement between hosts. If any client accidentally goes offline, the server will block because TCP has been designed to guarantee that any data sent does get to its intended destination.
2) If any client becomes momentarily unavailable, the server will block for the duration of this period.
3) Significant system resources go into making the protocol reliable over wide area networks, including the internet.
4) Because TCP is a stream socket, applications making I/O calls must make special effort to read or send all requested bytes since the function calls do not guarantee 100% results. A call to read 1400B may return 960 B only on the first call. Unless a second call is made, the remaining 440Bytes remain in the system buffer. This situation arises because TCP ensures that segments do arrive in sequence, so the remaining 440 which might have gotten lost , may be retransmitted with the next 1400B data segment sent. The result is two calls to read 1400B each may actually require three calls to read data segments of sizes 960 B, 440B and 1400B.
5) TCP has features like flow control and congestion control that are necessary for wide area networks like the internet but are not needed for the task at hand. This is because a LAN will not contain routers subject to and requiring data flow control.

To use TCP for the application in this thesis, it must be made to run in a mode such that it does not block even if any client momentarily slows down or goes offline. To this end, a non-blocking model of TCP was explored. Using TCP in non-blocking mode ensures that function calls like to send and receive data return immediately on the server side without waiting. The server must also be made to discard data when its system buffer fills up with a backlog of unsent data due to a slow client. Otherwise the

host TCP will block the server from generating any more data and this will definitely interfere with the server's primary function. Another drawback to the connection-oriented protocol is that it does not support a one-to-many connection. For the server to be able to publish data to more than one client, it has to establish a separate connection for each client. Though this is not impossible to accomplish, it reduces bandwidth since all the clients will share the same bandwidth and data must be separately transmitted even though the data is identical. Therefore for N clients, the available bandwidth to any one client will be 1/N the total bandwidth.

This bandwidth reduction issue is easily solved in UDP using a UDP broadcast. User Datagram Protocol supports one-to-many communication but UDP has its own limitations. Just as the connection-oriented nature of TCP poses a challenge in implementing TCP sockets for the task, so does the connectionless nature of UDP. Sockets based on UDP are prone to errors due to lack of monitoring of transmission success. There is no guarantee that a datagram sent across the network will ever get to its destination. However, thanks to the checksum calculated on the header and data, any datagram that gets to its destination is expected to be free from corruption. Because UDP does not incur as much overhead as TCP, it is usually faster than TCP and easier to implement.

Unlike TCP that creates virtual connections between pairs of applications, UDP treats every single communication as an independent task. Each datagram sent is treated independently of other datagrams. Because the protocol does not exchange state information concerning the sockets, the server cannot know whether the published data ever gets to the intended destinations. This is a mixed blessing in the sense that, while the situation makes UDP an unreliable transport protocol, it also means there is no interference of server operation by client activities. Because UDP does not guarantee delivery, any application that uses the protocol must solve the issues of reliability itself. It is therefore desirable to make UDP more reliable by introducing mechanisms similar to those under TCP, without necessarily recreating TCP. Methods of improving the reliability of UDP considered in this thesis include:

1) a hand-shaking approach similar to TCP acknowledgement (ACK),
2) worker threads with data buffering on the server side,
3) non-blocking mode of UDP sockets,
4) broadcast of data for one-to-many communication.

The UDP approach has been implemented on an embedded network device, the Netburner SB72IO-300 as a case study. The SB72IO-300 10/100BaseT Ethernet Device is a module that solves the problem of network-enabling devices with 10/100 Ethernet. Powered by a 32-bit high-performance Motorola ColdFire 5272 processor, the board enables design engineers to design network control and monitoring functionalities into their applications without the need to run these applications from PC. The SB72IO comes fully loaded with a TCP/IP stack, has on-board 8MB SDRAM for buffering and running of applications.

For evaluating the real-time performance of TCP and UDP sockets, programs were developed in C/C++ that implemented the simple but standard TCP and UDP clients and the server. Results obtained from evaluating the TCP sockets showed that the TCP server could be blocked if the client delayed in receiving the published data. While the simple UDP sockets did not block, data delivery from the server to the client(s) was not guaranteed.

To prevent the simple TCP server from blocking, the software was modified to include a worker thread that was responsible for maintaining the client-server virtual connection. This effectively insulates the main server application from adverse effects of this connection. The result proved that the TCP client no longer could block the improved TCP server. The program code was also written such that the improved TCP server could handle multiple clients at the same time.

In a similar way, the simple UDP client and server applications were modified. The client software included a worker thread that processed the received datagrams, thus allowing the main application to be able to detect and make requests for retransmission

of datagrams that it loses. The simple server software was modified to include a worker thread that processed all retransmission requests from the listening clients. When the real-time performances of the improved server and client programs were evaluated, they proved that the improved client applications perform better than their standard clients, in an instance reading 100% datagrams at rates up to 8 MB/sec when the simple client could read 100% only at rates of 4 MB/sec and lower. Unlike the improved TCP server, an improved UDP server used UDP broadcast to send datagrams to multiple clients with no reduction in bandwidth.

The thesis work has been divided into 4 chapters. Chapter 1 gives brief background information on basic internet concepts, how the TCP/IP protocol suite was developed and what services are available in this suite with special emphasis being made on the transport service providers. A snapshot of the network setup used in the thesis is given in this chapter as well. Chapter 2 looks at the limitations of the protocols in achieving the set goal. Chapter 3 is dedicated to finding methods of making TCP as non-interfering as possible with respect to the data publishing system. Chapter 4 covers UDP and techniques to make UDP more reliable without re-inventing another TCP. Finally, a conclusion on how and when to use the various approaches is given in the last portion of the work.

CHAPTER ONE

1.0 BACKGROUND

1.1 TCP/IP PROTOCOL SUITE

Networking and internetworking has been primarily motivated by the need for sharing resources and information among computers, irrespective of their physical location. Computer systems communicate on a given network following rules and signals defined by protocols. Design and implementation of network protocols is a difficult task and is usually made simpler by subdividing into smaller tasks or layers.

Early in the history of communication, the International Organization for Standardization (ISO) defined a layered model to help protocol designers have standard designs that could serve communication needs. It was important that designs were standardized so that arbitrary pairs of computers could communicate without concern for the kind of communication systems to which they are connected. All the layers together in the model make up what is referred to as a Protocol Suite or Protocol Stack. Among the number of protocols designed for the internet, the most popular and most widely used is the Transmission Control/Internet Protocol (TCP/IP) suite of protocols. The Defense Advanced Research Projects Agency (DARPA) originally designed TCP/IP for use by the US military for data communication [2].

In figure 1.1, the services provided by the various layers have been briefly indicated. The layers are designed such that each layer communicates with adjacent layers through a well-defined interface. Figure 1.2 shows the TCP/IP suite built on the OSI 7-layered reference model. Communication between applications on the same computer flows down the stack from the application layer to the interface layer, then up to the peer application. If communication is taking place among applications on different computers on the network, then data flows from one application layer down the stack, onto the network and then up the peer's stack on another computer. As shown in figure 1.3, two

Application	Specify how applications use the network.
Presentation	Specify how to represent data.
Session	Specify how to establish a communication session with a remote system.
Transport	Provide details on data transfer.
Network	Specify formats regarding packet addressing and forwarding in a network.
Data Link	Responsible for organizing data into frames and transmitting the frames over a network.
Physical	Gives detailed specification of LAN hardware.

Figure 1.1 The OSI 7-layer reference model.

Application	Application
Presentation	
Session	
Transport	Transport
Network	Internet
Data Link	Interface
Physical	Physical

Figure 1.2 Comparison of the OSI and TCP/IP models.

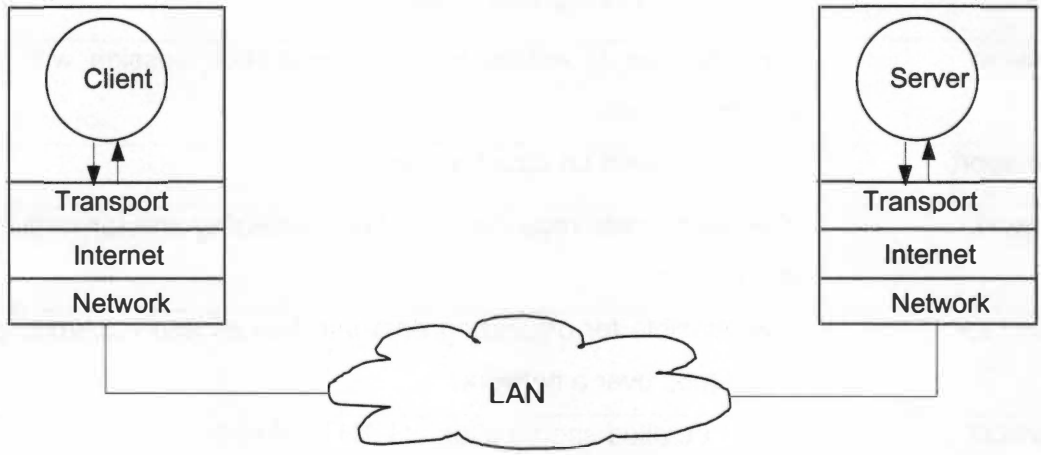

Figure 1.3 Virtual link between client/server applications.

applications communicate with one another through a virtual link established over the LAN by the transport protocols.

1.2 THE TRANSPORT PROTOCOLS

The transport protocols are among the most complex of the protocols in the TCP/IP suite. These protocols deal with end-to-end communication over the hardware and the software of the network. The TCP/IP suite employs two transport protocols in data communication: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

Transmission Control Protocol is a reliable host-to-host protocol used between hosts in packet-switched networks. It is very robust and reliable with the following characteristics:

1) Connection-oriented : There exists a virtual connection between TCP connected hosts.
2) Full duplex : allows two-way communication.
3) Guaranteed delivery service between TCP peers.
4) Has control over data flow: A client receives only as much data as it has buffer space for.
5) Congestion control: A server sends as much data as the system can handle without intermediate router overload.

The last two points, though useful in the Internet which spans the whole world, is not relevant in a LAN which does not involve the use of routers. Routers are often bottlenecks in the Internet since they serve as communication links between networks. All the above characteristics are possible because the two host computers do have a virtual connection and constantly exchange state information. This virtual connection plays an important role when implementing the client-server problem in this thesis as explained in the chapter on TCP socket implementation.

The UDP is a connectionless message-oriented best-effort protocol with the following characteristics :

1) Connectionless: No connection is established before communications between hosts begin.
2) Full duplex.
3) No guarantee that a UDP packet (called datagram) will get to its destination.
4) No control over data flow.
5) No congestion control.

Not establishing a connection between hosts makes the implementation of UDP sockets easier but less reliable. Any application programmer who decides to use UDP for communication must solve issues arising from lack of guaranteed delivery of UDP datagrams, even if the client and server are located over a network the size of a LAN. Experiments conducted on UDP sockets show that datagrams are lost even if the client and server are located on the same local host [10].

1.3 LOCAL AREA NETWORK

A Local Area Network is a computer network that uses technology designed to span a small geographic area. Local Area Networks are characterized by low propagation delays and high transmission rate, limited number of computers that can be connected as well as limited distance over which the LAN can span. In a LAN, users connect to one another via cables, radio waves and/or other media, sharing data and other resources such as printers. This means the cloud-like figure labeled LAN in figure 1.3 can be expanded to show a number of computers and other peripheral devices all connected together by a medium such as coaxial cables . The network used in this thesis work is shown in figure 1.4.

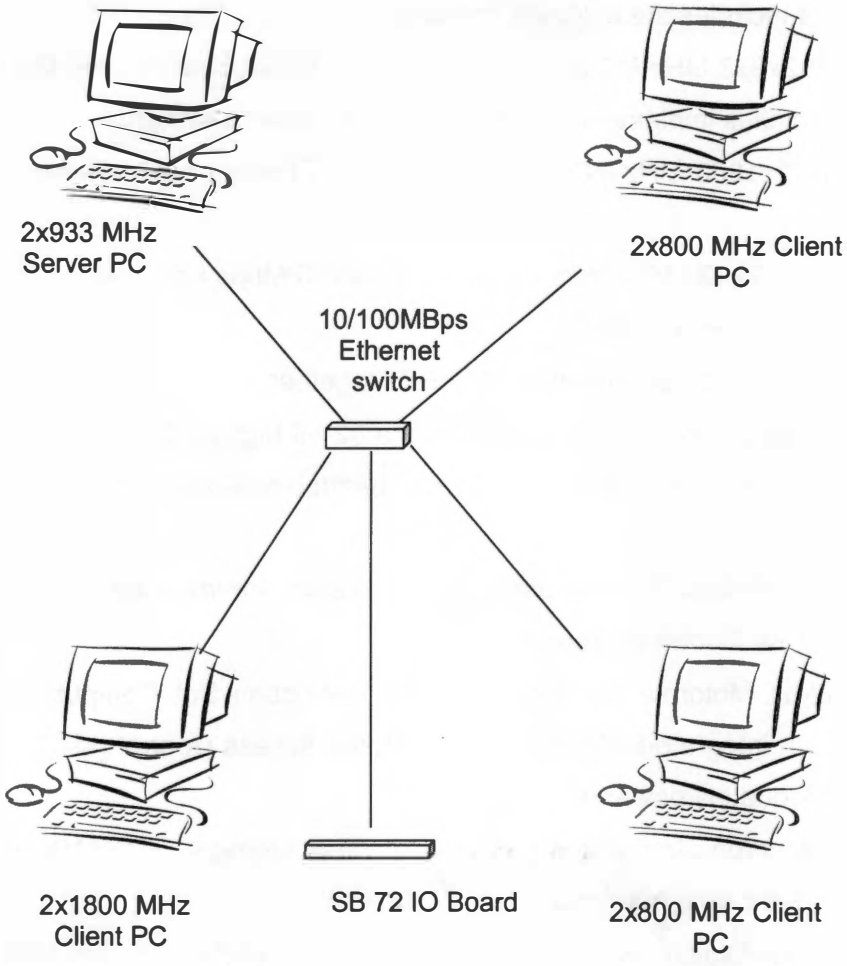

Figure 1.4 Topology of LAN used in experiments.

1.4 HARDWARE SETUP

The LAN in figure 1.4 includes the following hardware:

- A Pentium III 2x933 MHz PC with 10/100 Mbps Ethernet card running the server application which is implementing the embedded real-time system
- A Pentium IV 2x1800 MHz PC with 10/100 Mbps Ethernet card running a client application.
- Two Pentium III 2x800 MHz PCs each with a 10/100 Mbps Ethernet card serving as additional monitoring stations
- A 10/100 Mbps switch for connecting the PCs together.
- A dedicated board SB72IO-300 powered by a 32-bit high performance Motorola 5272 processor as another example of an embedded real-time system.

The dedicated board SB72IO-300 was used as a case study for implementing the UDP algorithm because of the following useful features:

- Powerful 32-bit Motorola ColdFire Reduced Instruction Set Computer (RISC) processor with integrated 10/100 Ethernet Media Access Control (MAC) makes the board adequately powerful.
- 8 Mbytes Synchronous Dynamic Random Access Memory (SDRAM) provides enough space for data buffering and applications
- 8 10-bit Analog/Digital inputs make it possible to implement real-time data acquisition.
- Real Time Clock allows application performance to be timed.

1.5 SOFTWARE SETUP

Software for implementing and evaluating the TCP and UDP sockets was developed in C/C++ and compiled with Microsoft Visual C++ 6.0. All PCs are running under Windows 2000 Professional. The dedicated board is run under uC/OS real-time operating system and its application is written in C/C++ .The C/C++ programming language was chosen because it is a standard language that offers excellent execution performance.

1.6 MODELING A REAL-TIME DATA ACQUISITION SYSTEM

In practice, a computer may be taking data from a device such as a Peripheral Component Interconnect (PCI) board that has a controller on it. A typical example is a PCI Data Acquisition (DAQ) board as developed by UTSI. This PCI board is controlled by an AMCC PCI Matchmaker S5933 controller with a maximum data transfer rate of 132 MB/s. The PCI board does Direct Memory Access (DMA) transfer from the board to the computer periodically. DMA is a technique for transferring data from a device to main memory without passing the data through the Central Processing Unit (CPU). The computer usually has two data buffers that it switches between when receiving data from the PCI DAQ board. The computer first enables a DMA transfer into one by supplying the address and size of memory to the DAQ system. Following this, the computer waits on the DAQ system to do the DMA transfer into the buffer indicated. The DAQ system then sets a bit that indicates that data is ready for processing. The computer then enables buffer number two and starts processing on data in buffer number 1. Upon completion, it waits on the DAQ to set the bit again signifying that data in buffer number two is ready. The computer then enables a DMA transfer to buffer one memory and starts processing data in buffer two. The sequence of the computer enabling a DMA transfer into a buffer, processing data in another buffer, and waiting on the DAQ to set a bit is repeated continuously.

While waiting for the DAQ system to place data into the computer's data buffer, the operating system of the computer may require CPU time for something else. Depending on the data transfer rate of the DAQ system and the length of time that the CPU is called away, a DMA transfer may be over when the computer returns to wait on the DAQ system. At this point the DAQ system is waiting on the computer to enable the next buffer and so cannot transfer data. While waiting, the DAQ stores data in its onboard memory. As soon as the computer enables a DMA transfer into the next available buffer, the DAQ quickly transfers this backlog of data. The computer at this point is lagging behind its schedule in this processing function and therefore processes data as quickly as possible. The computer may be able to catch up with the DAQ system depending on its processing power and DAQ transfer rate. This means the

backlog of data on the DAQ memory may remain constant or may start to decrease with time after a temporary backlog buildup.

Rather than use an actual DAQ board with a typically fixed data rate, in this thesis the board was simulated using the time stamp counter available on Pentium and later PCs. The time stamp counter is a counter which increases a 64-bit count at the rate of the computer CPU speed. Supposing the CPU speed is F Hz, then the time stamp is increased every 1/F seconds, or the time stamp counter counts F cycles every second. The time stamp is reset to zero when the computer is powered on, or when the counter increases it to 2^{64}. This maximum cannot be reached even within the lifetime of the PC at the frequency of 933 MHz because it will take

$$= \frac{2^{64}}{(\text{days in a year})(\text{seconds in a day})(\text{CPU Speed in Hz})} \quad (1)$$

$$\approx 627 \text{years}.$$

In simulating the server application with a specified desired data rate, a DMA transfer is ready every N_C cycle counts calculated as

$$N_C = \frac{(\text{packet size in Bytes})(\text{CPU speed in MHz})}{(\text{desired data rate in MB/s})}. \quad (2)$$

If C_{last} is the cycle count when the last data packet was sent, it is time to send another packet at C_{next} given by

$$C_{next} = C_{last} + N_C \quad (3)$$

Just as the DAQ does a DMA transfer every specified time interval to the computer, the server application at every N_C time count publishes data on the LAN. If the server gets behind the DMA transfers, a backlog is generated and this is calculated in MB as

$$backlog = \frac{(C_{actual} - C_{next})(\text{packet size in B})}{(N_C * 1000000)} \quad (4)$$

where $C_{actual} > C_{next}$.

This backlog therefore represents the backlog of data generated on the DAQ onboard memory when the CPU got interrupted. The value of this backlog therefore should never exceed the amount of memory available on the DAQ board since this will cause the DAQ system to fail in its critical function as a real-time data acquisition system.

1.7 BASIC SOCKET API

Applications communicate with one another using the transport protocol through an interface as previously seen in figure 1.3 on page 8. The interface is known as the Application Program Interface (API) and it contains rules and procedures that the application software uses to access the service. Although there are many APIs existing, the Socket API (abbreviated as sockets) is the most widely used. The minimum socket API calls needed by a client and a server applications are shown in the block diagram in figure 1.5. there are a few variations depending on the type of socket created: UDP socket or TCP socket.

Beginning with the client application, the first task is to obtain a socket for communicating with the server. This is achieved by the socket() function call. The application next calls *connect()* to establish a TCP virtual connection or to create an association with another UDP host. The client is then ready to receive data from the network system through calls to *recv()* or *recvfrom()*. It is important to note that the connect() function call is optional with UDP sockets since the *sendto()* call does implicit association of the local host with the remote host. At the end of the communication session, the application calls the *closesocket()* function for the network system to release allocated system resources. A server application goes through a slightly different path as shown in the diagram. After obtaining a socket, the server application binds the named socket to a local address and port with a call to the function bind(). Following this action, the server can now send data with the *send()* or *sendto()* calls. The same *closesocket()* function call is made as the client in order to allow the system to release allocated system resources.

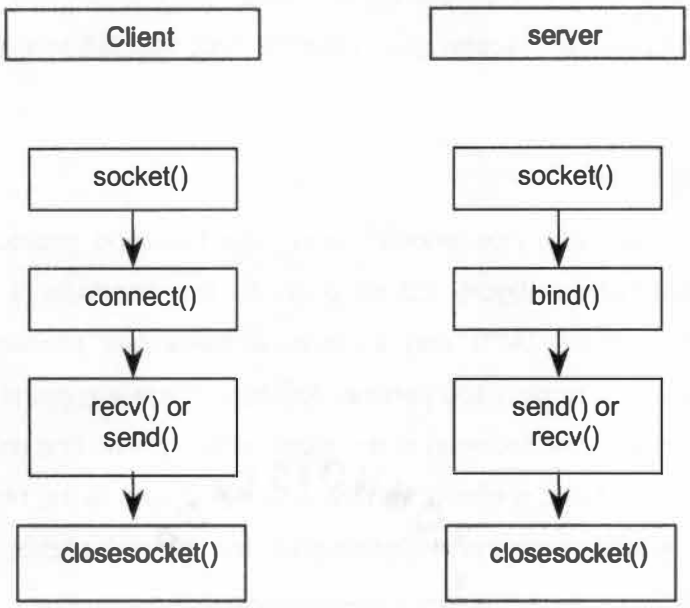

Figure 1.5 Basic socket calls in network applications.

CHAPTER TWO

2.0 LIMITATIONS OF AND EVALUATION OF THE PROTOCOLS

2.1 TCP SOCKETS

Due to the nature of TCP sockets, it is necessary to consider the following points :

1) TCP requires time to establish and terminate each connection. Over a long period, this becomes insignificant and may be overlooked in overall performance.
2) No response from a client means a server will wait in a blocked state monitoring the connection.
3) Most importantly, connection-oriented means there cannot be any easy one-to-many relationship without reducing bandwidth.

The second point constitutes what is termed as failure modes of TCP when used in a real-time application. These occur whenever the virtual connection between TCP peers get broken. The conditions under which this occurs include :

1) A TCP client crashing.
2) A network outage.
3) A peer TCP crashing.

In the first instance, the client is terminated before the end of transmission. The peer TCP notifies the server that the client is unavailable with a FIN signal and thus drops the connection. Software was developed in C/C++ for evaluating the real-time performance of standard TCP sockets. When this was simulated on a Pentium 3 2x933 MHz computer, the CPU usage went up from 19 % to over 51% in the period that the server discarded the send efforts. The increase in CPU usage is a clear indication of an undesirable influence on the server. On the other hand if the application simply blocks and does not respond to data received with ACKs, the server TCP will time out and retransmit the data. This continues until the sending TCP drops the connection with an error. A network outage is another case where the virtual connection gets broken. Just

as in the previous case, the server drops the connection after timing out. In BSD TCP/IP stack, this occurs after approximately 9 minutes [10]. The last mode is when the peer's TCP crashes. In this state the peer's TCP is not able to inform the server's TCP of the dead connection so the server maintains the connection in a blocked state. Clearly all these conditions are undesirable and must be avoided in the design of a real-time system. A mechanism has to be devised so the server can detect the state of connections and decide what to do with the connections.

TCP has a mechanism that, when enabled by an application, is able to detect dead connections. Keep-alive is a special TCP segment that the TCP layers exchange periodically. If the peer TCP crashes the sending TCP notifies the server after series of probes have yielded no response. By default, the connection is dropped after more than 2 hours. Changing this value is usually possible on system-side and this affects all other TCP connections, thereby defeating the purpose of the keep-alive function for all other applications [10]

2.2 REAL-TIME PERFORMANCE EVALUATION OF A SIMPLE TCP CLIENT AND SERVER

As explained in section 1.2, TCP creates a virtual connection between the client and server applications. This connection allows the client to have a direct influence on the performance of the server. This section will demonstrate a qualitative measure of the influence of the client on the performance of the server by putting the client in a momentarily blocked state. The client could be blocked because the operating system is busy doing something more critical or because the client application is locked up doing something very intensive such as writing data to the hard disk. In such a situation, until the present task is completed, the client will not be able to execute the next instruction.

The effect of a blocked client, or at least a client that is temporarily unavailable, on a server has been simulated in the simple TCP example. The simple TCP example involved developing codes in C/C++ that implemented standard TCP sockets. In the

example, the client is an application that uses a TCP socket to receive data. The client is made to simply receive TCP segments and determine the percentage of the total bytes received at the end of transmission. The flow chart of the client software is shown in figure 2.1.

The server is made to loop through *NLoops* sending TCP segments to the client. The server does this by periodically checking the time stamp counter of the PC to see if it is time to send data. As explained previously, the time stamp count is a function of the CPU speed and is the total number of clock cycles that the processor has executed since the time it was powered on. Using equation (2), let N_C be the number of clock cycles after which the server must send data. If C_{last} is the cycle count when the last segment was sent, then it is time to send another segment at C_{next} given by equation (3) as $C_{next} = C_{last} + N_C$.

If the server happens to be delayed and more than N_C clock cycles have elapsed since the last segment was sent, a backlog of data has accumulated in an intermediate board. If C_{actual} is the actual count when the server checks the counter ($C_{actual} > C_{next}$) then this backlog in MB is given by equation (4) as

$$backlog = \frac{(C_{actual} - C_{next})(packet\ size\ in\ B)}{(N_C * 1000000)} \qquad (5)$$

The factor *packet* size represents the size of a TCP segment. Each segment has been chosen to be 1400 B as a figure close to the Ethernet frame size, 1500 B. For communicating with the TCP client application, a software was developed for the TCP server and the flow chart of this application is shown on figure 2.2.

In evaluating the real-time performance of the client and server, the client was made to delay for 5 seconds after sending 30,000 segments. During this period, the server continues to send data until the client's receiving buffer and the server's own sending

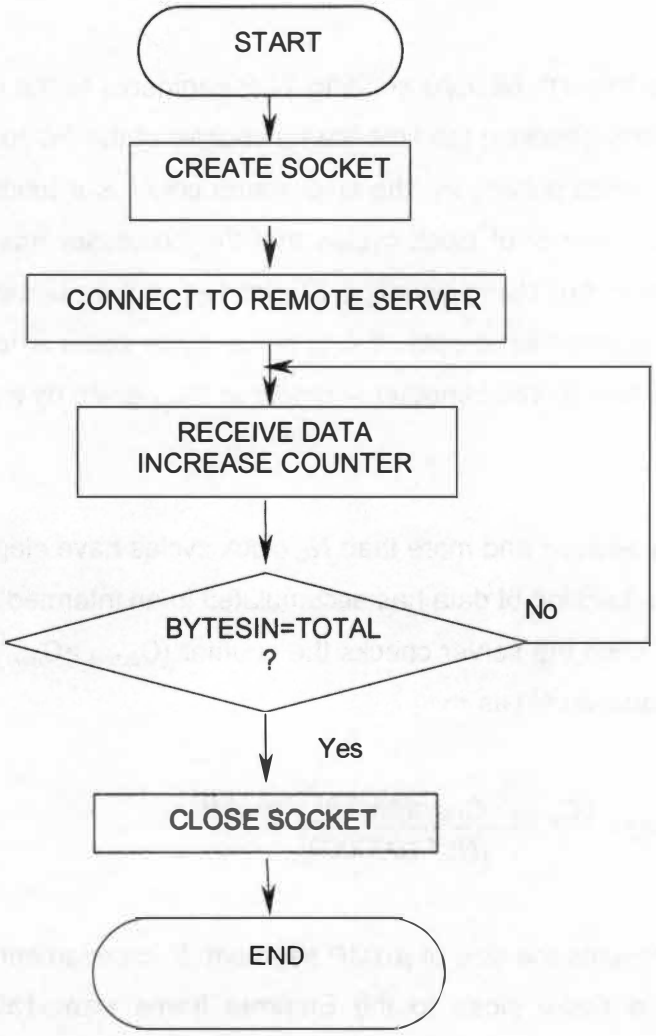

Fig 2.1 Simple TCP client flow chart.

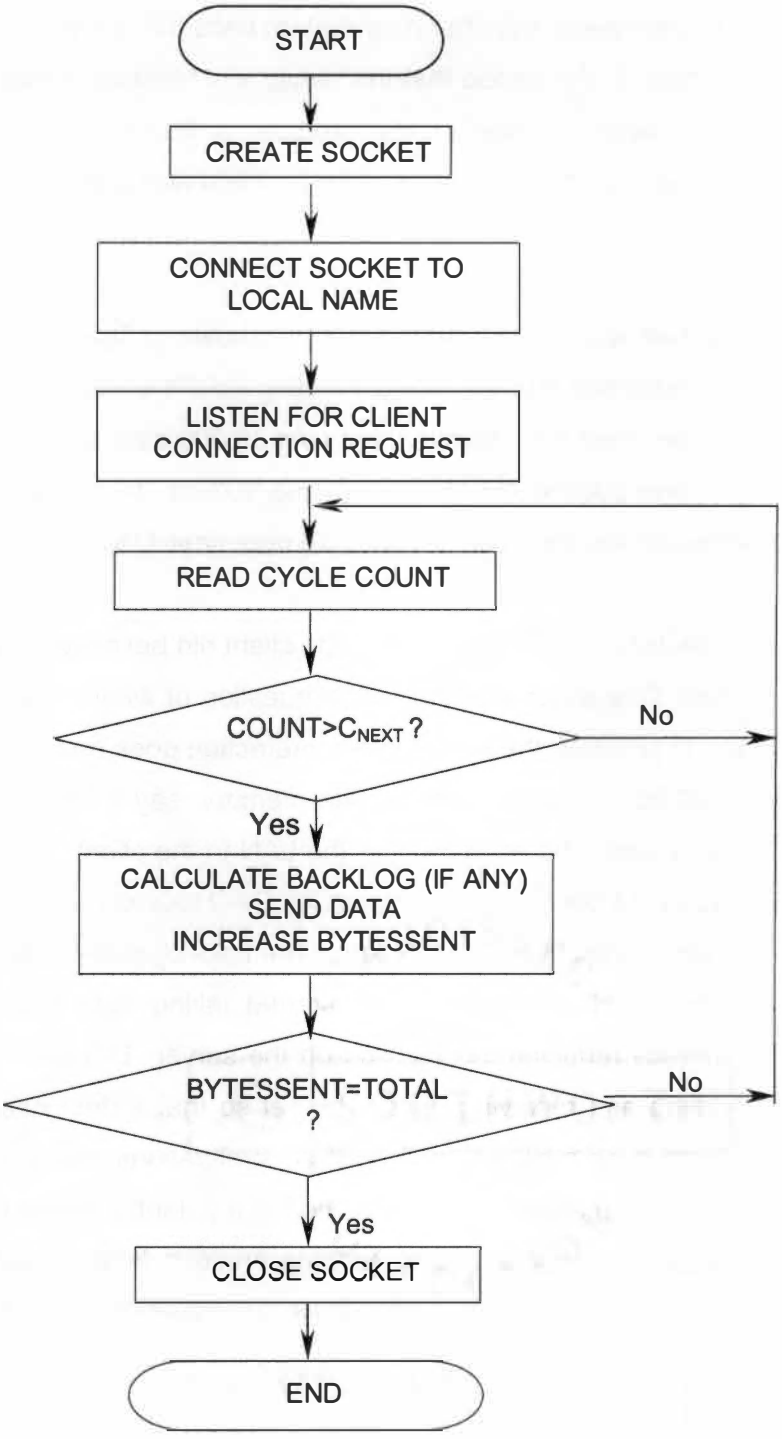

Figure 2.2 Simple TCP server flow chart.

buffer get full. When this happens, the operating system halts the server until the client starts to request more data. In the period that the server sits blocked, it falls behind its sending process so that when it resumes the process, a nonzero backlog can be calculated which gives a quantitative indication of the problem with a simple TCP socket approach.

The result when the server was sending at 8MB/sec is shown in figure 2.3. The initial portion of the plot is flat indicating that the rate of sending data is being matched by rate of receiving data. Then the client paused after receiving 30,000 data segments. As was explained above, after filling both receiving and sending buffers, the server paused 7 s after beginning transmission and resumed somewhere near time 11s.

The server did not pause for exactly 5 seconds as the client did because it was sending data into the two buffers. This action introduces the question of what happens if these buffers are made large. In practice, the server-client interaction goes on continuously as the server transfers data from a device with a finite memory, say 8 MB as it is in the case of the SB72IO-300 board, and sends it over the LAN to the client. Since it cannot be determined how much data backlog is built up on the DAQ memory, a backlog on the server greater than its set threshold is to be avoided. The backlog in the example above started decreasing after the client resumed and started taking data faster than the server was sending, thereby reducing the backlog on the server. The question then is how large can the sending and receiving buffers be set so that a data backlog never happens ? Certainly there is no such figure. Therefore in situations where the backlog increases during a transmission serves as an indication of a potential server failure. The solution to this problem lies in separating the sending process from the process that transfers data from the board. This approach has been developed in the chapter under TCP implementation.

Performance of the server as the rates are varied is shown on the figure 2.4. Since this backlog increases with time, the server application cannot send data at rates above 11.8 MB/sec without failing.

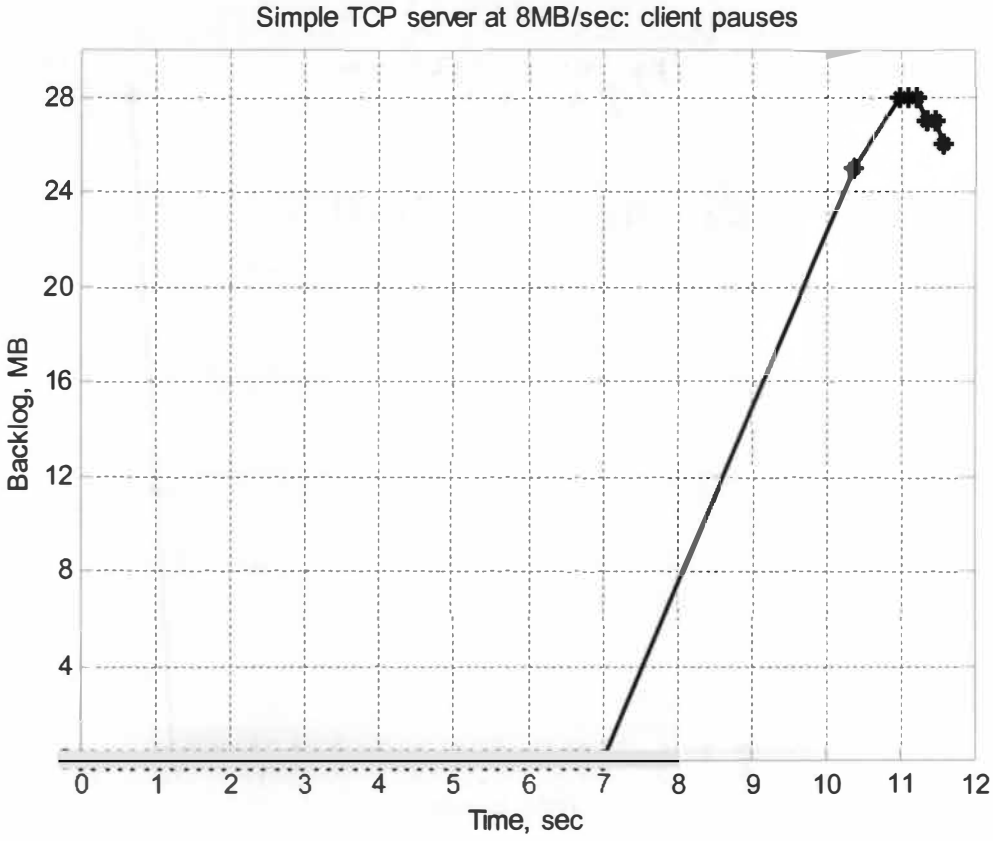

Fig 2.3 Performance of a simple TCP server as client blocks.

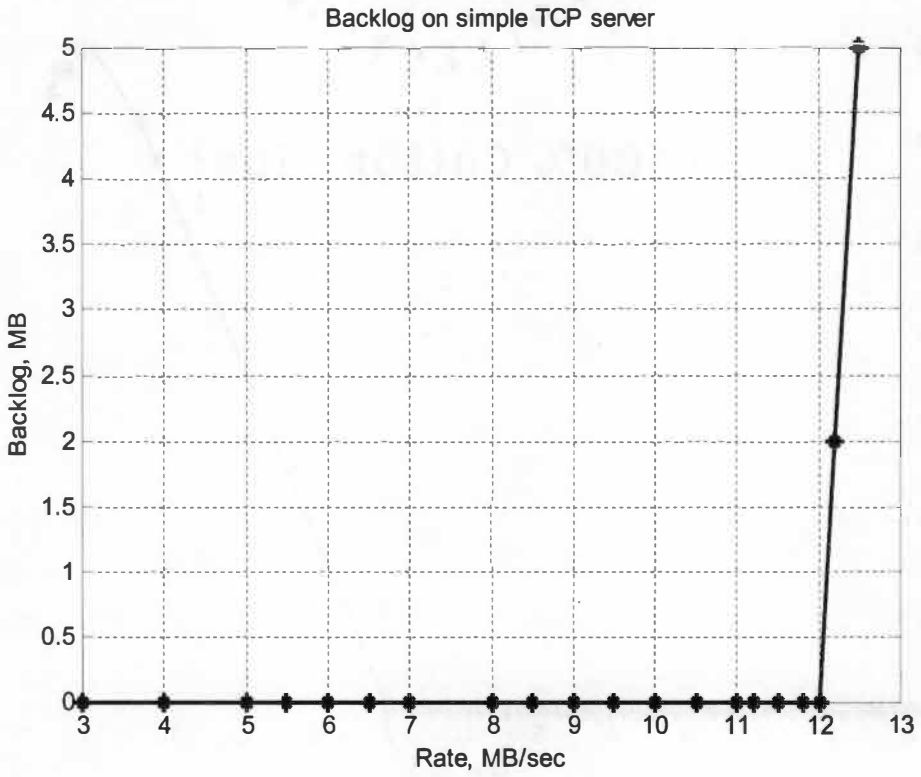

Fig 2.4 Backlog on simple TCP server as transmission rate increases.

2.3 UDP SOCKETS

User Datagram Protocol does not establish a virtual connection between peers and there is no exchange of state information between peers. Each transaction is considered independent of others before or after it. Therefore applications using UDP transport services must have mechanisms to ensure the required reliability. The first effort in using UDP sockets is to ascertain how well the protocol performs.

In terms of data integrity, the system performs a checksum test on both header and data so any datagram that gets to its destination is guaranteed to be free of errors. Because UDP is a message-oriented protocol, it is not likely to have a partial datagram received, especially since the size of a datagram was chosen to be 1400 Bytes, which is smaller than the Ethernet frame size of 1500 Bytes. Therefore there are no datagram fragments produced in this or any of the tests carried out.

2.4 REAL-TIME PERFORMANCE EVALUATION OF A SIMPLE UDP CLIENT AND SERVER

For evaluating the real-time performance of simple UDP sockets, two applications were developed in C/C++ and tested. Flow charts for the UDP client and server applications are shown in figures 2.5 and 2.6 respectively. In this test, the server goes through *NLOOPS* to send 1400 MB of data to the client. Periodically the server checks the time stamp count and decides if it is time to send the next datagram. After sending the datagram, it increases the sequence number (SEQNO) and checks if it has counted through the NLOOPS. The client simply counts the number of datagrams received and finds this as a percentage of the total expected number of datagrams. All the system buffers are set to 7 MB for the same reason as for TCP, that is in order to improve performance. Even though a bigger value may improve performance, it also increases the chance of the system call silently failing. By default, the system buffer is set at 8KB so if the system is unable to provide the requested value, it returns not the next immediate smaller value,

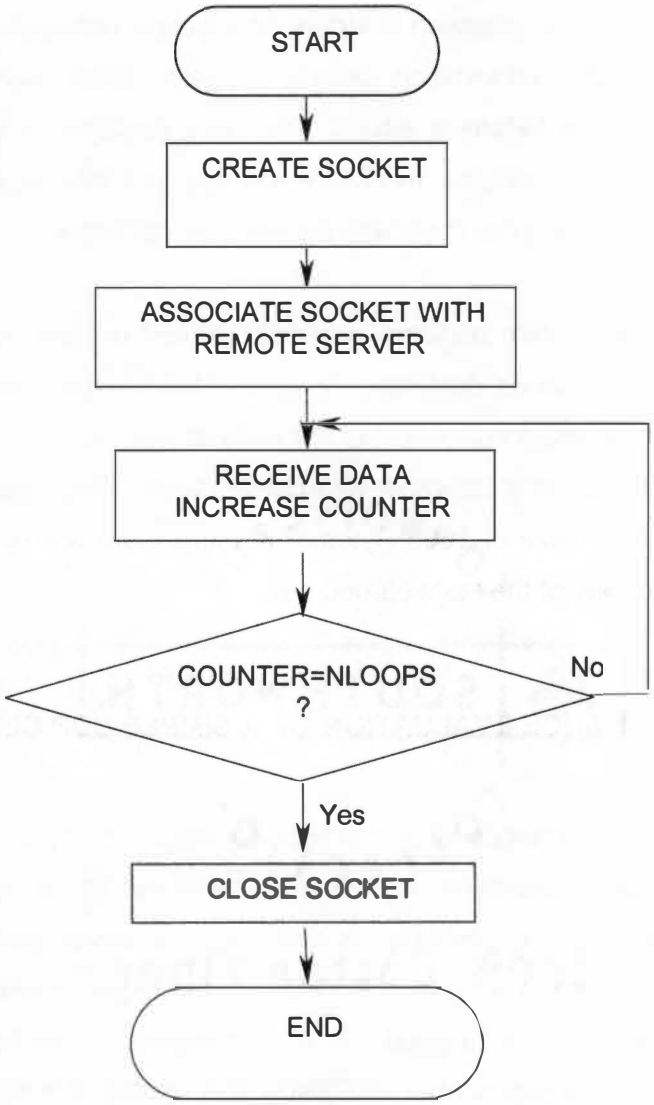

Figure 2.5 Simple UDP client flow chart.

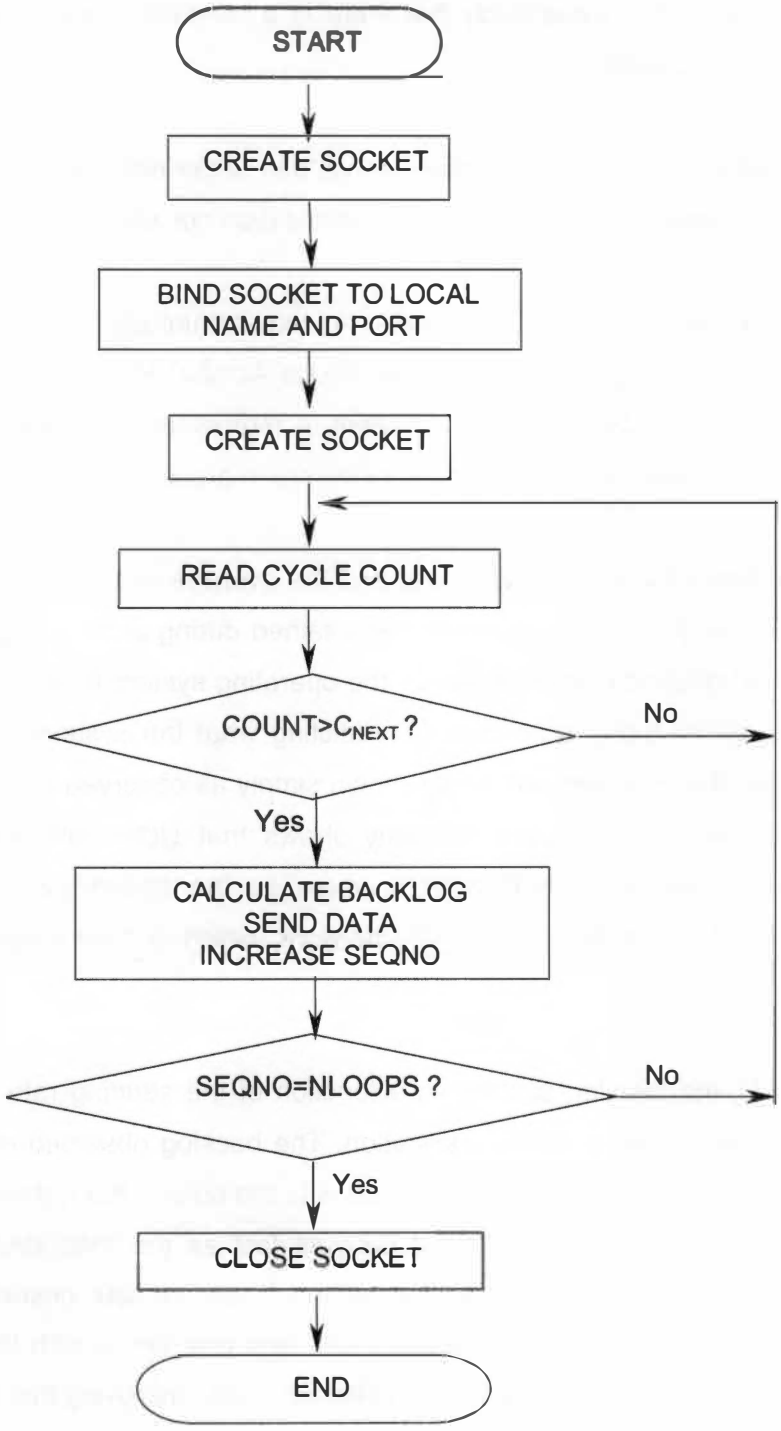

Figure 2.6 Simple UDP server flow chart.

but the default value. Setting a value such that there is a relatively good chance of success is considered appropriate.

The test was performed for the simple UDP client-server pair under two circumstances :

1) Idle : the client pc was not running any other application nor was it used to load anything.
2) Not Idle : In this case the client PC was made to load and unload programs such as Visual C++ 6.0, Internet Explorer 5.0 and Adobe Acrobat Reader 5.1. These served as momentary disturbances to the client. A momentary disturbance was used to demonstrate how unreliable UDP sockets are in practice.

The test was run five times for each sending rate and the average taken. Despite this, there is no guarantee that the same results will be obtained during another test since the performance of UDP depends largely on what the operating system is doing during the period of the test. Since there is no way of predicting what the system could be doing at any given time, these results are being shown simply as observed behavior of the UDP sockets. The result in figure 2.7 clearly shows that UDP truly does not guarantee data delivery, and the situation gets worse as the transmission speed approaches the threshold of the Ethernet 10/100 network, which is theoretically 12.5 MB/sec.

As was case with TCP, the backlog of data as a function of the sending rate is also used to determine the performance of the application. The backlog observed in figure 2.8 was not due to the inability of the client to keep up with the server, but rather due to the fact that the system could simply not send data as fast as the theoretical 12.5 MB/sec speed of the Ethernet. Thus the backlog begins to accumulate beyond 11.8 MB/sec rate. Not much can be done about this but the main problem is with the poor performance of the client, especially when the PC gets disturbed. Improving this is what has been looked into in chapter 4.

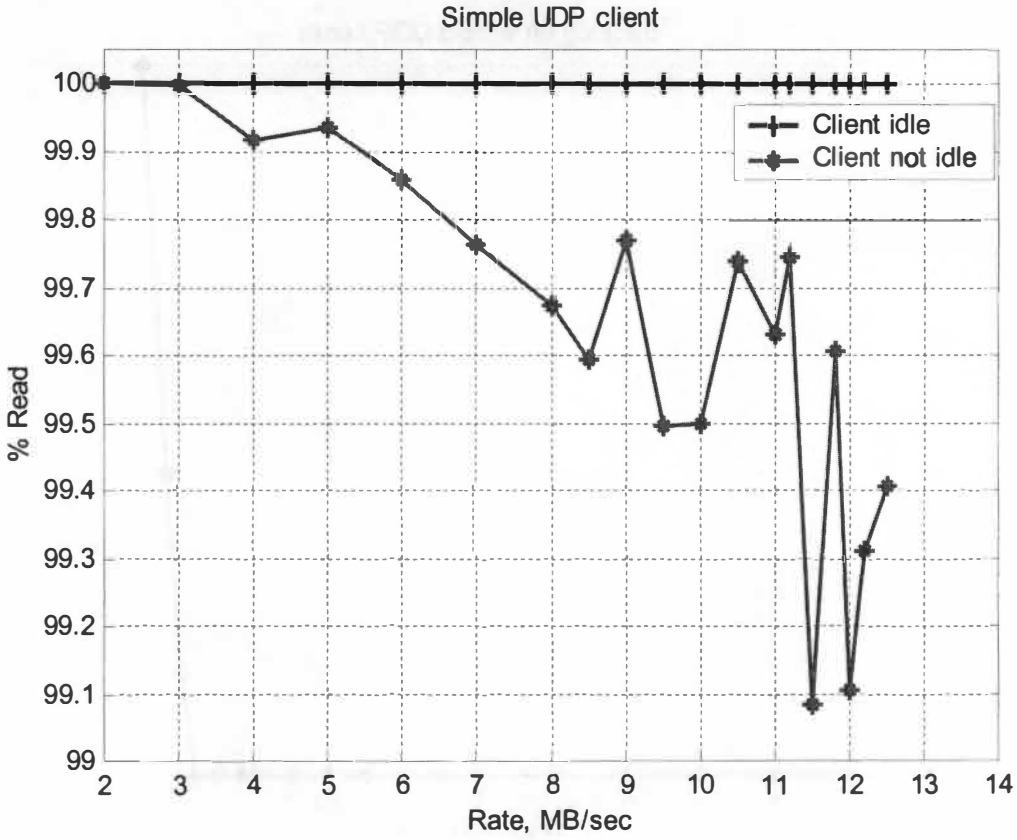

Figure 2.7 Performance of simple UDP client when PC was idle and when PC was disturbed.

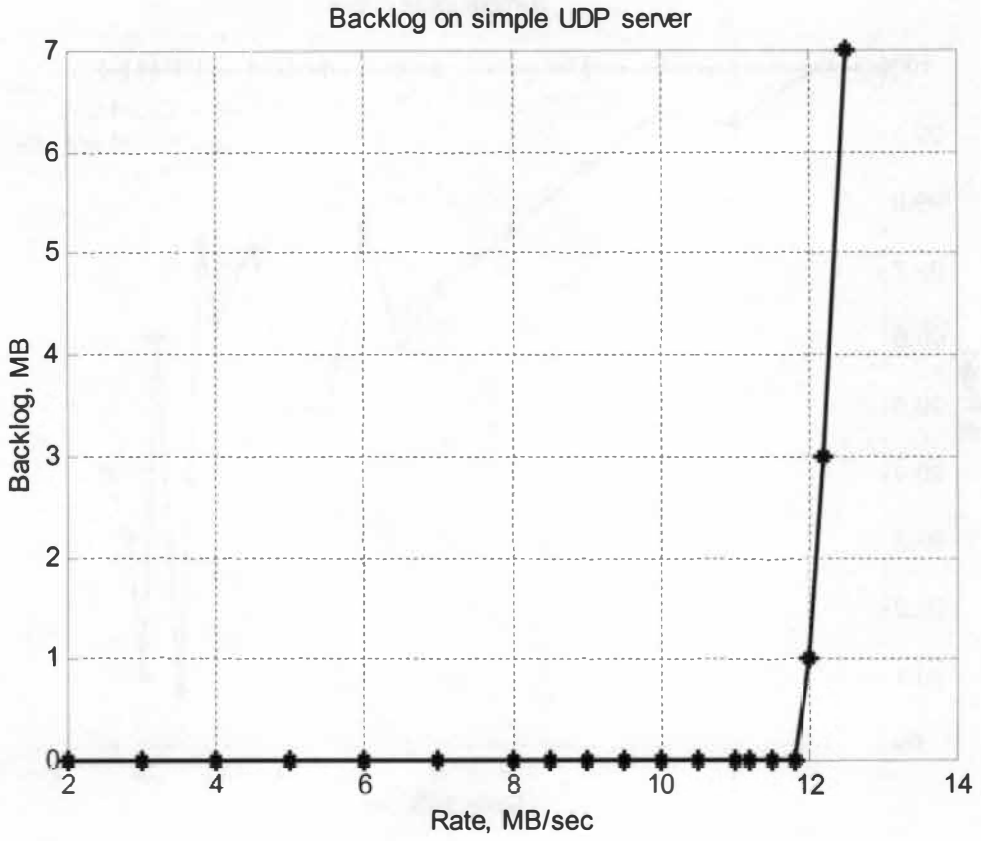

Figure 2.8 Backlog on simple UDP server.

CHAPTER THREE

3.0 SOLUTION TO THE TCP PROBLEM

3.1 TCP SERVER WITH WORKER THREAD

The main concern is to prevent the server from blocking whenever the client ceases to respond. It should be possible to prevent this blocking by putting the procedure that sends data into a separate thread. This would ensure that anything that goes on in that section will not directly affect operation of the main program. Software was developed in C/C++ that implemented this approach according to the flow chart shown in figure 3.1. As figure 3.1 shows, the sending procedure is located in the worker thread.

The worker thread waits for the main thread to provide valid sockets for all the clients. The worker thread then waits for data to be put into the data buffer, that is it waits until the data buffer count (BUFFCOUNT) is greater that zero. The worker thread sends this data, decrements the data buffer count and then increments the segment sequence number (SEQNO). While this buffer count is greater than zero, the worker thread will work on sending data to the client. In this mode of operation, data backlog will only accumulate if this data buffer gets filled up. Therefore the bigger this buffer, the less likely there will be backlog as was experienced in the case with the simple server. The main thread is responsible for taking care of connecting all clients and sending the sockets used over to the worker thread to use. The main thread periodically checks to see if there is data to be put into the data buffer. If there is data ready, it writes the data into the data buffer, increments the BUFFCOUNT and calculates the backlog if there is any. Putting the sending procedure in a separate thread allows the main thread to run without discarding data until its data buffer fills up.

The results shown in figure 3.2 are from a test carried out with the client which stalled in its data receiving process after receiving 30,000 data segments as was done in the test with the simple TCP server-client. The result of the backlog is shown on the plot in figure 3.2.

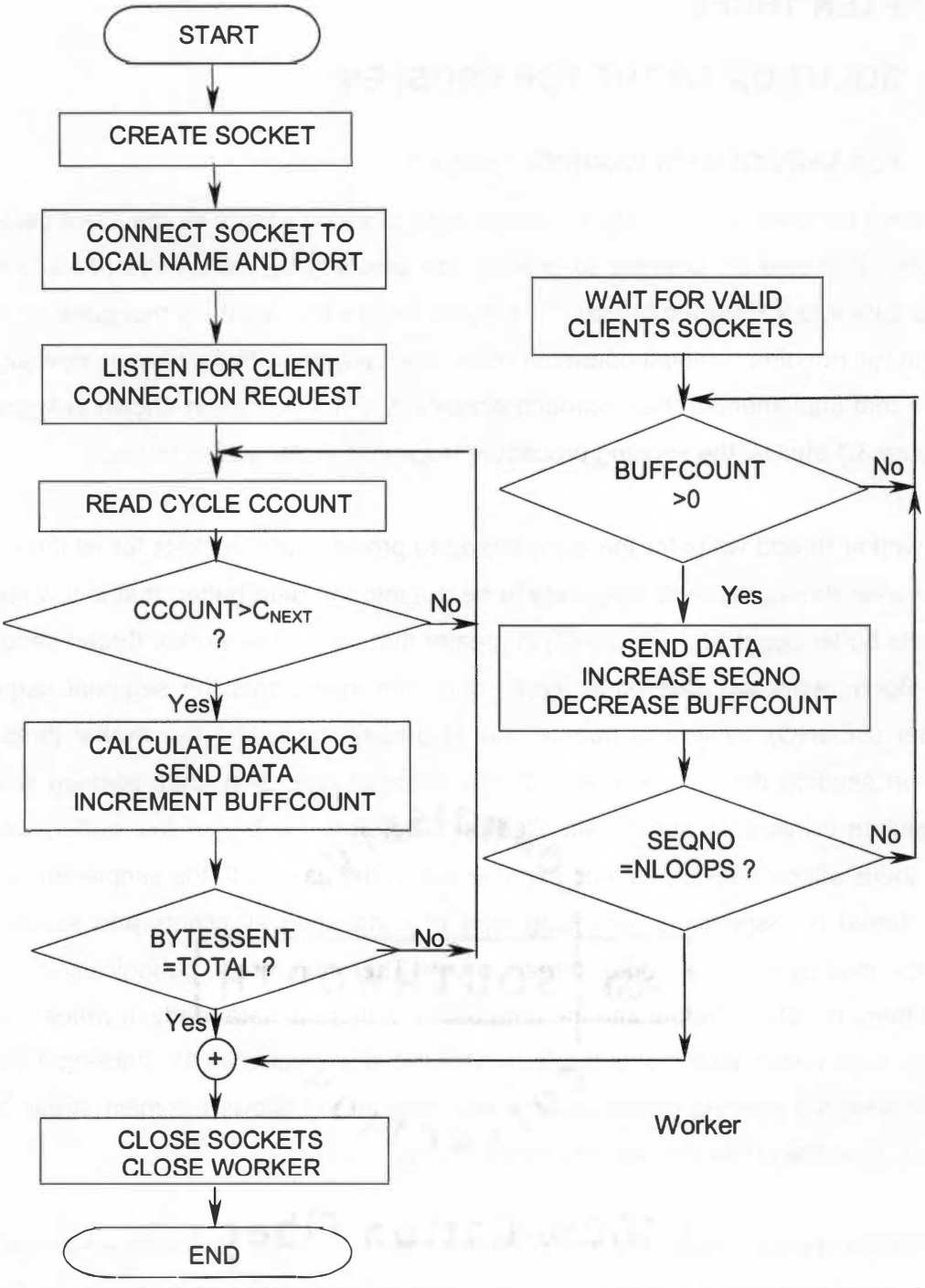

Fig 3.1 Flow chart of the TCP server with worker thread.

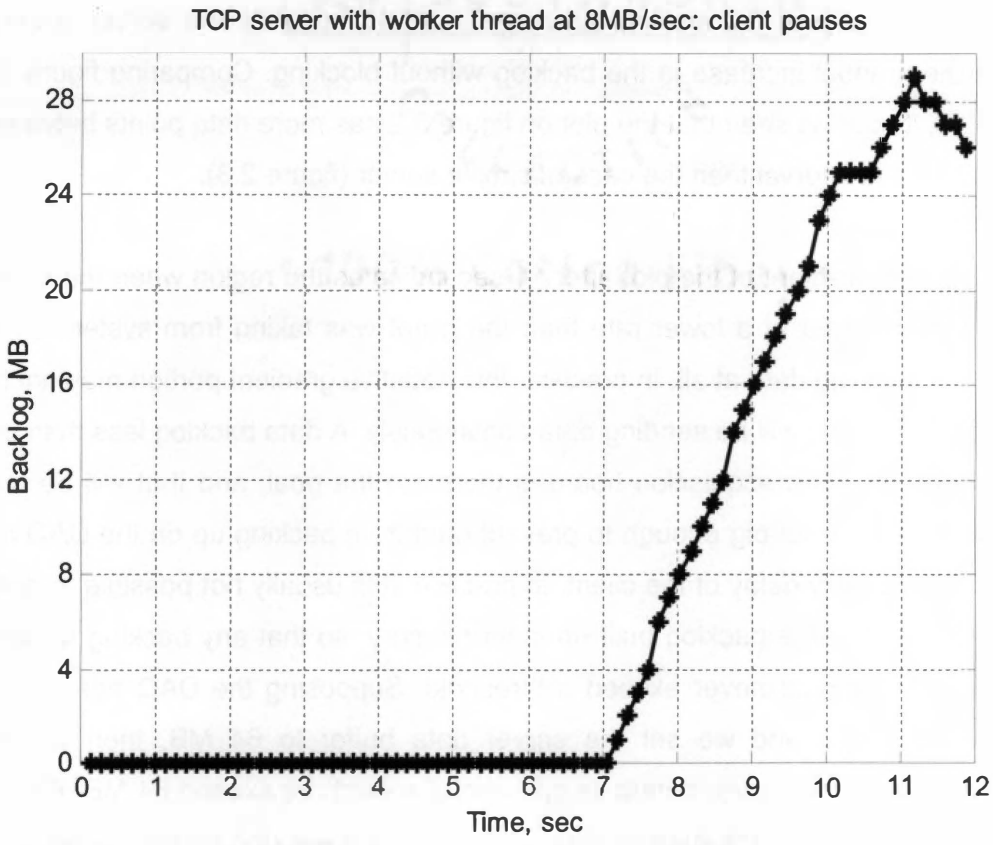

Figure 3.2 Performance of TCP server with worker thread.

As the plot of figure 3.2 shows, between 7 and 11 seconds the server continued to record the gradual increase in the backlog without blocking. Comparing figure 3.2 with figure 2.3, it can be seen that the plot on figure 3.2 has more data points between the 7 sec and 11 sec interval than the case of simple server (figure 2.3).

The negative gradient of the plot after 11 sec indicates the region when the server was either sending data at a lower rate than the client was taking from system buffers, or simply not sending data at all. In practice, this negative gradient portion may not happen because the server will be sending data continuously. A data backlog less than installed memory on the data acquisition board is therefore the goal, and that will be achieved with a data buffer set big enough to prevent data from backing up on the DAQ memory due to a temporary delay of the client. In practice, it is usually not possible to determine when the DAQ has a backlog built up in its memory, so that any backlog accumulated on the server should never exceed a threshold. Supposing the DAQ has an onboard memory of 8 MB and we set the server data buffer to 64 MB, then the backlog accumulated on the server during its operations should not exceed 64 MB. Beyond this value, the server discards data so that at all times, the backlog on the server is like the first 7 seconds of the plot on figure 3.2. This is the recommended operation mode for the server because it prevents data buffer overflow on the interface board and allows the server to perform its time-critical processing.

To improve system performance, the sending and receiving system buffers of the sockets were set at 7 MB. This value was chosen because some of the boards developed at UTSI have 8 MB memory chips installed. Using exactly 8 MB may not leave room for the application being run. A typical example is the dedicated IO board SB72IO-300 that has 8 MB of SDRAM for data buffering as well as running applications. It is worth adding that a bigger sending or receiving buffer value also increases the chance of the call silently failing, that is not returning the requested value without returning a SOCKET_ERROR message. There is therefore no guarantee that the operation system will allocate all the requested memory.[8]

3.2 SERVER WITH MULTIPLE CLIENTS

Another limitation with TCP is caused by its connection-oriented nature. It does not support a one-to-many connection, meaning for data to be monitored by more than one monitoring clients on the LAN, additional separate sockets will have to be created for each station. All the connections then share the available bandwidth. Since this reduces the available bandwidth for transmission, any attempt to transmit at speeds higher than possible creates backlog and will cause the server to fail or discard some of the data going to the clients. In this work, three and two clients were tested and the results are shown on figure 3.3. The server uses Windows event notification to detect and connect all the clients that will be monitoring the published data. The sockets are put into an array and the worker thread uses a loop to send the same copy to each client. So for N clients, a given copy of data will be sent N times. This reduces the bandwidth for N clients by N fold. Included is server performance when the number of clients is three. As was to be expected, the backlog at a given rate increases with increase in number of clients. To prevent this from happening, at rate *R* MB/s, only *N* clients can be served such that

$$N \leq \frac{12}{R}. \tag{6}$$

This then is the maximum number of clients that can be served simultaneously by the TCP server when more than one client is needed to monitor the published data on the LAN. A backlog in excess of the server's data buffer is undesirable because the server will have to discard data that is being sent to the clients. To prevent this situation from occurring, the above equation must be adhered to at all times.

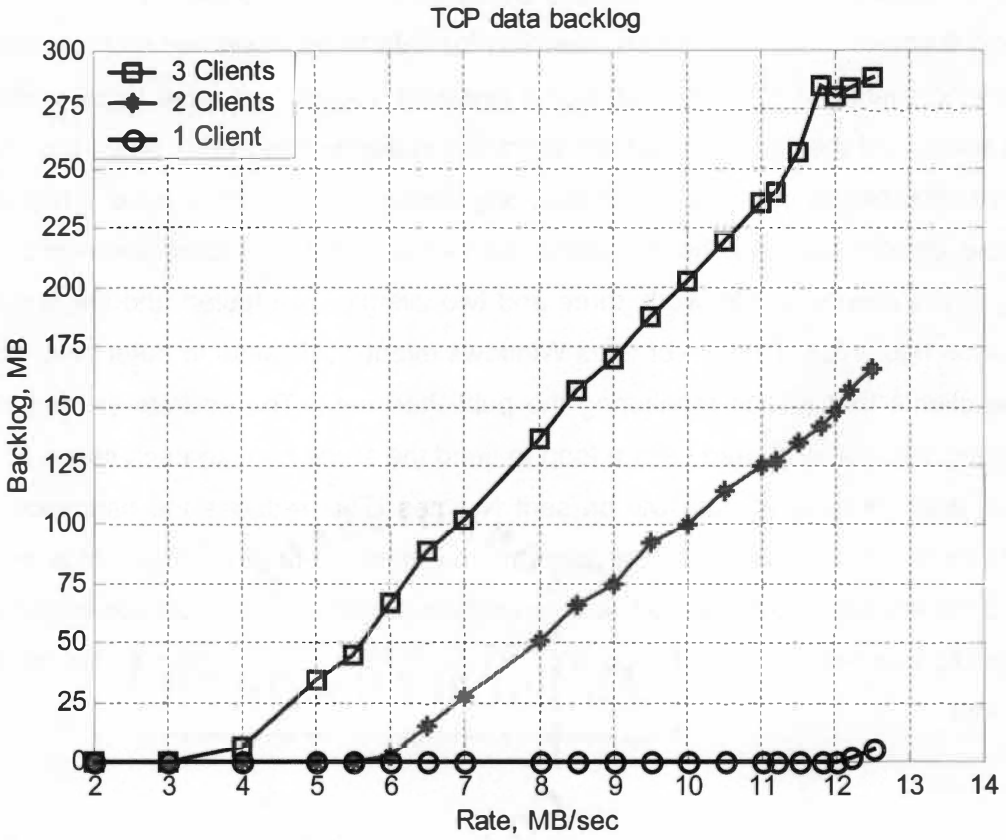

Figure 3.3 Performance of improved TCP server with multiple clients.

3.3 ANALYSIS OF RESULTS

3.3.1 Interference of improved server due to backlog

Backlog is a serious issue with TCP applications because of the fact that the protocol makes great efforts to prevent data loss. Even with the worker thread approach used for the server to negate the effect of client activities on the server, the backlog generated due to a server-to-many-clients communication can be significant depending on the transmission rate. This can be seen from figure 3.3. and it makes using TCP for multiple clients unsuitable.

3.3.2 Ability to handle multiple clients

As explained in section 3.2, TCP does not handle efficiently multiple clients. This is because TCP needs to establish a connection for each client and send a copy to these clients even though the data is identical. This is a waste of bandwidth and makes using TCP one-server-to-many-clients communication unsuitable.

3.3.3 Reliability

TCP does guarantee delivery from one TCP peer to another TCP peer. It does not however guarantee that data given to a TCP peer will be received by the intended application since anything may happen to the link between the application and its host TCP. This was explained in detail in section 2.1.

3.3.4 Maximum number of clients supported by the improved server

Under the TCP implementation the maximum number of clients possible is determined by the rate of data transmission by the server on the LAN and is given by 12/R, where R is the rate of transmission in MB/sec and 12 is the maximum observed rate (in MB/sec) on the 10/100 Ethernet (refer to figure 3.3). This formula presumes that the clients are not heavily loaded with a task that slows them down so much that the TCP host will stall for lack of sending window.

CHAPTER FOUR

4.0 SOLUTION TO THE UDP PROBLEM

4.1 UDP SERVER WITH WORKER THREAD AND BUFFERS

To improve upon UDP performance, a mechanism is needed to reduce the number of datagrams that a client misses. One solution is for the client to detect the datagrams it loses and request retransmission by the server. The server can use a worker thread to listen for and process the retransmission requests. Software was developed in C/C++ for testing and evaluating the suggested UDP solution.

The flow chart for the developed client software is given in figure 4.1. Figure 4.1 shows that the client uses two sockets: local and remote. The local socket is used for receiving broadcast datagrams from the server while the remote socket is used for making retransmission requests through another port to the server. The server listens for requests on this separate port, which has been set apart for direct server-to-client communication. The client begins by setting the expected datagram sequence number EXPTDSEQNO to null and waits for the first datagram whose sequence number is null. The sequence number counts PACKNO is set to the sequence number of the datagram received. The two sequence numbers, EXPTDSEQNO and PACKNO are compared and if they are equal, then the client updates EXPTDSEQNO, data buffer counter BUFFCOUNT and writes the datagram into a buffer for the worker thread's use. If EXPTDSEQNO is bigger than the PACKNO, then the datagram that was received is a retransmitted datagram so the retransmission count RETNO is updated. On the other if the EXPTDSEQNO is smaller than the PACKNO then the difference indicates a sequence of one or more lost datagrams that is requested from the client through a single retransmission.

Notice that this approach only allows for one retransmission request for a given lost packet. It would be possible to make another effort to recover datagrams lost after a first retransmission effort. This would have a success rate dependent on what caused the

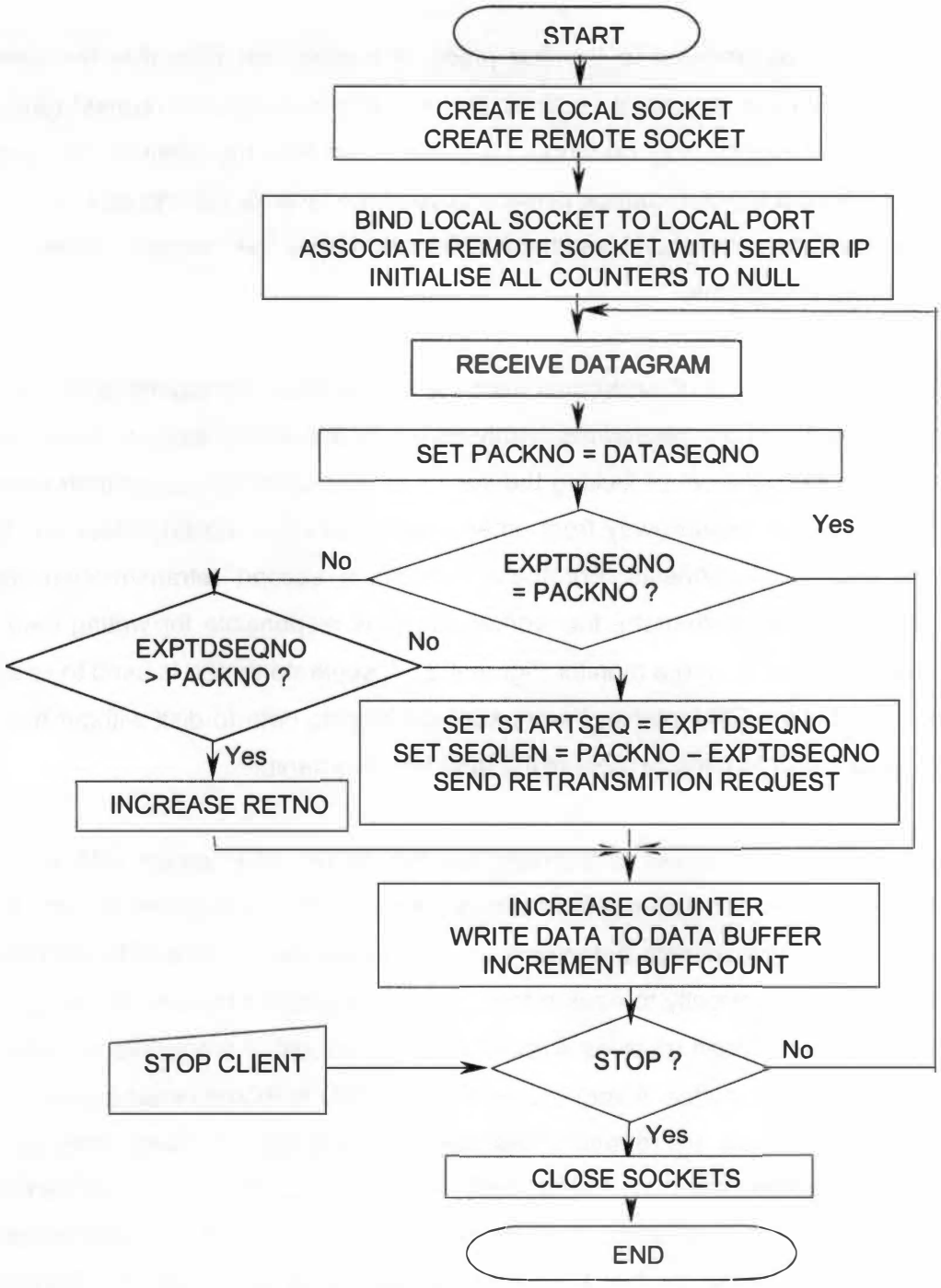

Figure 4.1 Flow chart of improved UDP client.

need for a retransmission in the first place. If a client lost data due to momentary interference, then it is possible that by the time a retransmission request gets to the client, this disturbance may no longer be present and thus the client will succeed in a repeated effort. If the disturbance persists beyond the time for making and receiving the first request, then a second retransmission effort may be needed to recover lost retransmitted datagrams.

If a client is losing 1% of datagrams from the server, then it is equally possible for the client to lose 1% of the datagrams retransmitted by the server. Second retransmission efforts have the potential of locking the server on one client for longer than necessary and may take bandwidth away from other clients that are expecting response to their first retransmission requests. For these reasons a second retransmission was not allowed. In the client diagram, the worker thread is responsible for writing data in the data buffer to disk or onto a monitor (figure 4.2). A separate thread is used to enable the client do the more CPU intensive work such as logging data to disk without negatively affecting its ability to capture most of the data from the server.

The improved UDP server is basically like the simple UDP server with a separate worker thread that processes retransmission requests from the clients (figures 4.3 and 4.4). The server broadcasts datagrams on one socket and uses a different dedicated socket for replying directly to each retransmission requests it receives from the clients. When the worker thread receives a retransmission request, it processes and stores the data into a request buffer. A request number REQNO is incremented signaling to the main thread that there are retransmission requests to be sent. Between checking on its time stamp counter, the main thread verifies that the REQNO is greater than zero. If REQNO is greater than zero, then the main thread responds to the request and continues on checking on the availability of data to be published. If retransmission efforts cause the server to lag behind its time count such that a backlog starts to build up, then when the backlog reaches a threshold, the server suspends servicing retransmission requests.

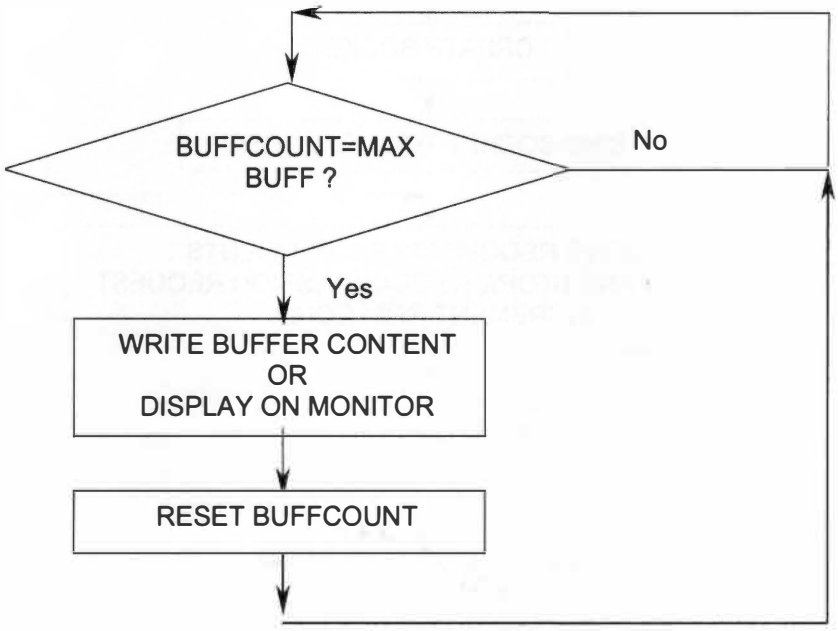

Figure 4.2 Worker thread for improved UDP client.

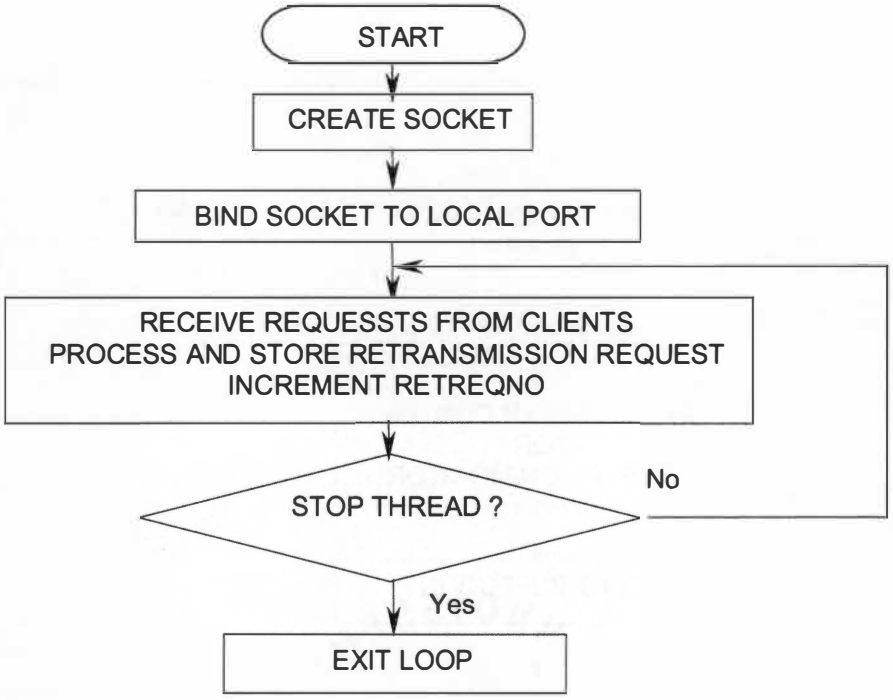

Figure 4.3 Worker thread for improved UDP server.

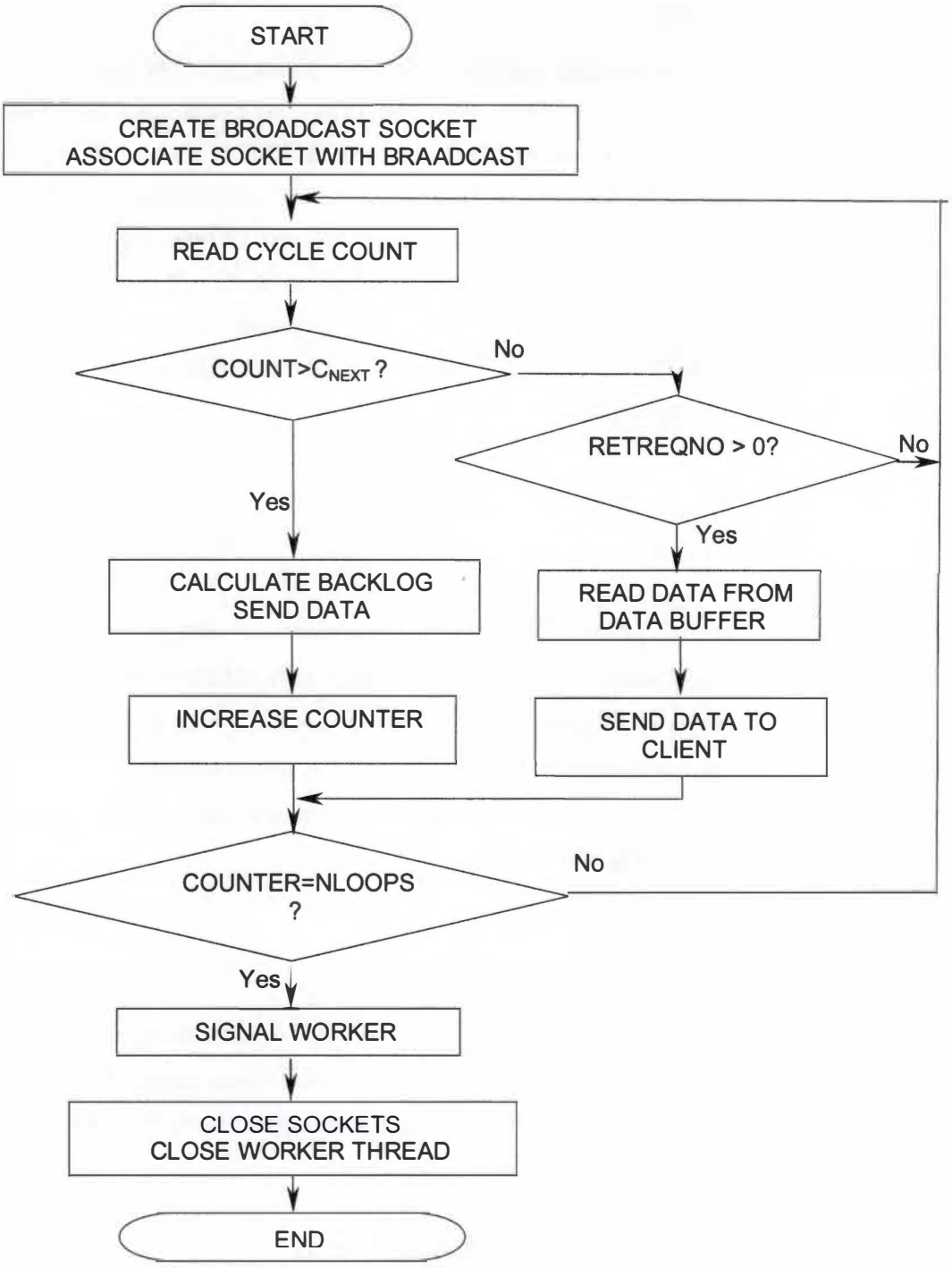

Figure 4.4 Flow chart of improved UDP server with worker thread.

4.1.1 Circular data buffer

To enable the server to retransmit datagrams lost by clients, it writes data sent into a circular data buffer. The data may be overwritten when the buffer gets full. The data buffer size is set big enough to hold 5000 datagrams, that is, set to 7 MB. Seven megabytes is chosen such that this algorithm could be implemented on dedicated devices such as the SB72IO-300 that have onboard memory of only 8 MB for buffering data and running applications. Implementing this buffer on a PC will require setting the value to a reasonable figure considering the available physical memory installed on the PC. At a speed of R MB/sec, it takes t_{buff} seconds to fill this buffer, where *databuffsize* is the size of the data buffer in MB.

$$t_{buff} = \frac{databuffsize}{R} \tag{7}$$

Theoretically, any speed below 12.5 MB/sec maximum throughput of the network allows room for retransmission to succeed. However, it was determined in practice that this limit is rather close to 11 MB/sec. Table 4.1 shows that at a rate of 11.0 MB/sec the server was able to use available bandwidth to service retransmission requests from clients. Therefore using 11 MB/sec in this calculation gives the minimum time to be expected in practice which is 0.636 seconds.

4.1.2 Request Buffer

Making room for more than one client, the server buffers retransmission requests from the clients. Observations have shown that whenever UDP loses datagrams, it does so for contiguous sequence of datagrams. As a result, the request buffer was chosen to be an array of records that include the following information

Start :	sequence number of first lost datagram,
Length :	length of the sequence,
IP :	IP address of the client making the request.

Table 4.1 Example of the number of retransmission datagrams requested by clients.

11 MB/s Client IP	Sequence No.	Length
192.168.0.10	8635	9
192.168.0.10	8676	18
192.168.0.10	11527	77
192.168.0.9	13591	54
192.168.0.9	13760	12
192.168.0.10	14111	38
192.168.0.9	17239	21
192.168.0.10	17251	26
192.168.0.10	19899	94
192.168.0.12	41241	224
192.168.0.12	41673	141
192.168.0.12	47268	811
192.168.0.12	58370	342
192.168.0.12	58989	355
192.168.0.12	64640	416
192.168.0.12	70351	416
192.168.0.12	76071	112
192.168.0.12	76184	1
192.168.0.12	76447	142
192.168.0.10	80895	62
	Average	169

To choose the size of the request buffer, the average round trip time for a request from a client had to be determined. Setting this buffer larger without a corresponding increase in the data buffer size is probably not useful; the slot for the requested datagram will be occupied by a new datagram since the data buffer is a circular one. The measured time between a request and a response was measured to be approximately 0.001 sec. This measurement was obtained by making a client request 10 datagrams from the server at 11.5 MB/sec and 8 MB/sec (refer to tables 4.2 and 4.3) The difference between the time stamp counts when requesting and when receiving these datagrams was measured and the roundtrip time $T_{roundtrip}$ is calculated as follows :

$$T_{roundtrip} = \frac{Received\_TimeStamp - Sent\_TimeStamp}{CPU\_SPEED}, sec \qquad (8)$$

The maximum among the two cases was chosen; this was approximately 0.00064 sec.

The time an operating system may take in receiving a datagram and sending it to an application is a factor that cannot be determined easily. It depends on how busy a machine may be at the time the datagram is received and also how powerful the processor is. Taking this point into consideration, the time between making a request and receiving the result was estimated.

In the 0.636 seconds that the data buffer rolls up, there can be theoretically a maximum number of requests from any client given by

$$\frac{0.636}{T_{roundtrip} + T_{client} + T_{server}} \qquad (9)$$

where T_{client} and T_{server} are respectively the times it takes the client and the server systems to detect and respond to the presence of a datagram in the system buffer. If T_{client} and T_{server} are both much less than 1 msec, then the denominator of equation (9) could be approximated to $T_{roundtrip}$, which is also approximated to 0.001 seconds.

Table 4.2 Example of the time between making and receiving a retransmission request at 8 MB/sec.

#	Time Stamp, when		Time (s)
	Received	Sent	
0	1387802272716	1387802086292	0.000372848
1	1387802587072	1387802450420	0.000273304
2	1387802874788	1387802709964	0.000329648
3	1387803215074	1387803110660	0.000208828
4	1387803476874	1387803339494	0.00027476
5	1387803735584	1387803631124	0.00020892
6	1387804062546	1387803858370	0.000408352
7	1387804402560	1387804296480	0.00021216
8	1387804661822	1387804525264	0.000273116
9	1387804953720	1387804848138	0.000211164
		Max	0.000408352

Table 4.3 Example of the time between making and receiving a retransmission request at 11.5 MB/sec.

#	Time Stamp, when		Time (s)
	Received	Sent	
0	1343375103402	1343374785112	0.00063658
1	1343375443312	1343375336976	0.000212672
2	1343375727520	1343375565254	0.000324532
3	1343376064400	1343375958422	0.000211956
4	1343376326394	1343376186922	0.000278944
5	1343376614952	1343376478390	0.000273124
6	1343376937666	1343376736786	0.00040176
7	1343377272976	1343377167744	0.000210464
8	1343377530072	1343377395010	0.000270124
9	1343377820842	1343377715602	0.00021048
		Max	0.00063658

The maximum number of retransmission requests would then be 636. 636 is considered moderate since a round trip time of 0.001 sec is actually higher than what was measured through the approaches mentioned above. Since setting the request buffer bigger than this may be unnecessary unless the data buffer is also made bigger, the request buffer size was set to 500 datagrams.

4.1.3 Maximum number of clients supported

Assuming an average of 169 datagrams as shown in table 4.1 have been lost each time a client makes a request , then the server can provide valid data for 5000/169, approximately 29 requests whenever available bandwidth would permit. This number could be from just a single client or from 29 different clients since the clients receive the datagrams simultaneously. If the datagrams to be retransmitted do not overlap in the data buffer, then it is possible that the possible number of clients to be served could be greater than 29. Despite the optimistic look on the number of clients supported by the server, it must be noted that the time taken by a server to detect data in its buffer T_{server} must be taken into account. If a server delays longer than estimated, the retransmitted data may not be what a client requested. In cases when observed data loses exceed the average of 169 datagrams, the data buffer may be increased with memory considerations being observed. For the above reasons, the maximum number of clients that this server can support is uncertain. However, it can be argued that this server can support a minimum of 29 clients within the given time limits discussed so far.

4.1.4 Performance of the improved UDP server and client

To test the performance of the improved client and improved server, a series of tests were run at rates from 3 to 9MB/sec. At these rates, the server does not experience any backlog and therefore transmits data at the set rate as shown on figure 4.5 and

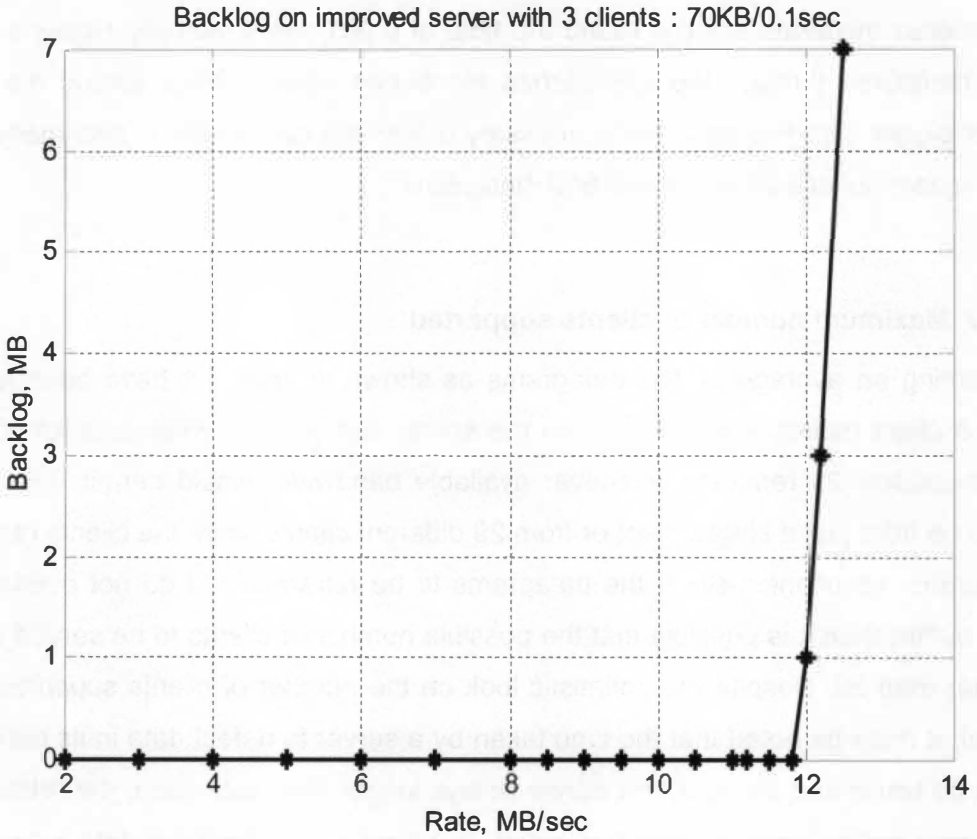

Figure.4.5 Toward determining limits of the improved UDP server.

table 4.4. The expected duration is compared to the recorded duration in table 4.4 to verify that the server is sending data at the desired rate.

The following three main tests were run :

1) Client PCs and applications were idle
2) Clients were writing data to disk at 700 KB/sec in two ways :
    a) writing a data buffer of 500 elements each second
    b) writing a data buffer of 50 elements each 100 ms
3) Improved server was tested with three improved clients and one simple client. This was to determine its performance under multiple clients.

To set an even platform for the test, the clients were made to write the data at a given rate and not as when they received data. This makes it possible to put all applications through the same number of disk write operations independent of the percentage of datagrams received. In cases where the performance of the simple and improved UDP clients was compared, both applications were run on the same PC under similar conditions. Figure 4.5 shows the results

The characteristics of figure 4.5 provides justification for not having these tests performed at rates above 9 MB/sec. Sending datagrams at rates between 2 and 9 MB/sec allows a 2.8 MB/sec margin for the server to operate without backlog. This is important because the main goal of the thesis was to prevent the data publishing from interfering with the time-critical functions of the server. Efforts to reduce data loss by clients through retransmission are suspended when this begins to cause deterioration of the server's performance. When the backlog drops below the threshold, retransmission effort is resumed. With zero backlog, the server is able to service more efficiently retransmission requests from its clients as table 4.4 indicates.

Table 4.4 Improved UDP server with three improved clients.

Rate	Backlog	Duration, sec		Requests	
MB/sec	MB	Recorded	Expected	Received	Sent
9	0	15.55864	15.55555556	11613	11495
8.8	0	15.91142	15.90909091	11947	11824
8.5	0	16.4708	16.47058824	10150	10150
8.2	0	17.07394	17.07317073	13665	13665
8	0	17.50018	17.5	6700	6688
7.5	0	18.66812	18.66666667	11860	11850
7	0	20.0001	20	13282	13282
6	0	23.3334	23.33333333	6553	6553
5	0	28.0001	28	5199	5199
4	0	35.0001	35	3149	3149
3	0	46.66672	46.66666667	203	203
2	0	70.0001	70	0	0

Figure 4.6 shows the performance of both simple and improved UDP clients when clients are idle. Following this test, two kinds of clients were tested on the second test where they both write 700 KB/sec to disk. The result of this test is shown on figure 4.7. For the simple UDP client this result contrasts sharply with the plot on figure 4.6 when both clients were idle.

Figure 4.7 clearly shows that the success of a simple UDP client depends on what the application is doing when the datagram is received. If the simple client is not involved in a CPU intensive activity, it performs very well as in figure 4.6. Figure 4.7 also shows the difference successful retransmission efforts make on the performance of the improved UDP client. While the improved client is more than 99.9% successful reading datagrams at the rates up to 7.5 MB/sec, the simple client gets this close only at rates close to 4 MB/sec.

The difference in performance even increases when the clients write the data in smaller blocks (refer to figure 4.8). The improved client performs better at rates 4 MB/sec higher. This wide difference is due to the fact that writing smaller blocks of data makes it less likely that the disturbance that caused the data loss will persist when the client receives the retransmitted datagrams.

The effect of different write buffer sizes is shown on figure 4.9 where the performance of an improved client is evaluated as it writes data to disk with 700 KB and 70 KB each sec and each 0.1 sec respectively. Even though the resulting rate is the same, the results show that it is better for the improved client to write data in smaller blocks.

Figure 4.10 demonstrates that in a group of clients served by the same server, the client that performs worst is the simple client which is unable to replace datagrams it loses, unlike its improved counterparts.

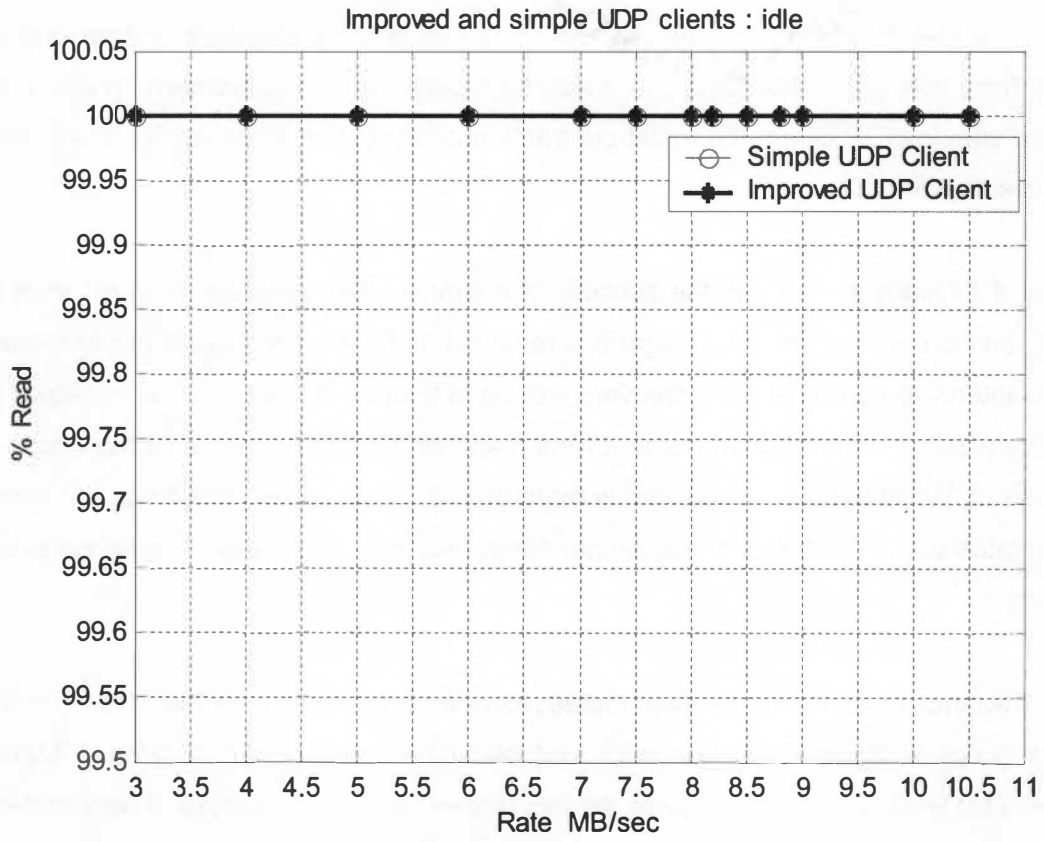

Figure 4.6 Comparing improved client and simple client when both are idle.

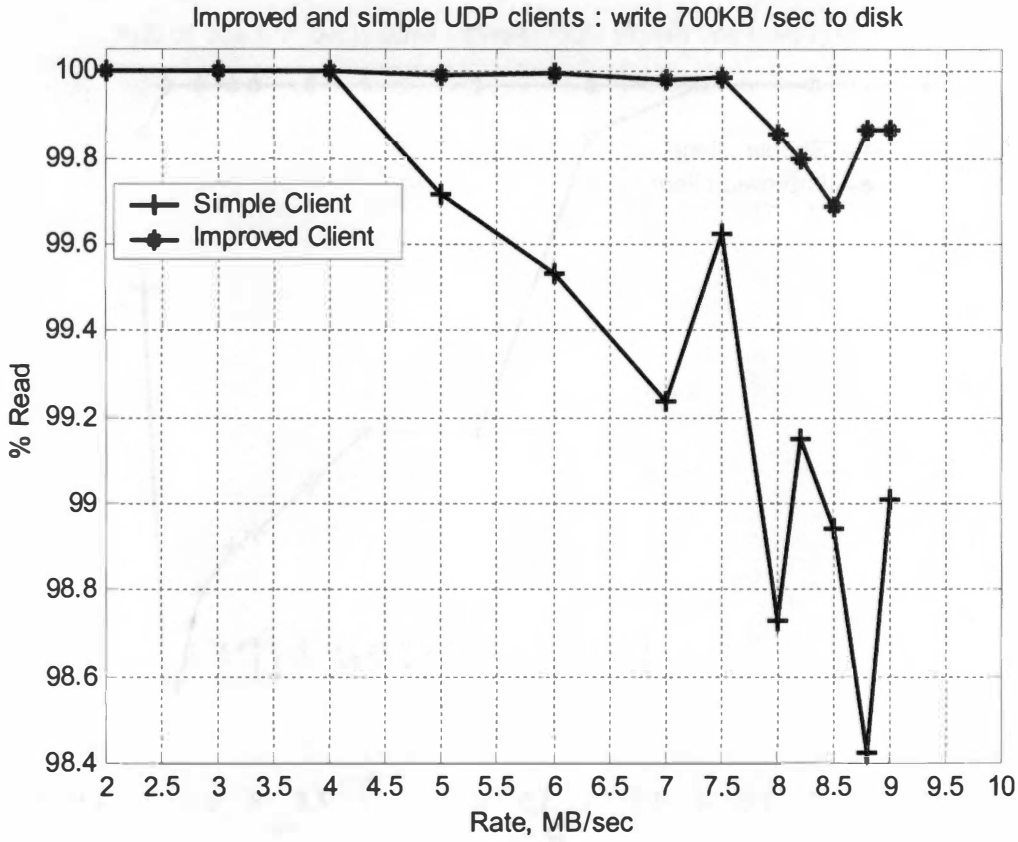

Figure 4.7 Comparing Improved client and simple client when writing to disk with a 500-datagram data buffer.

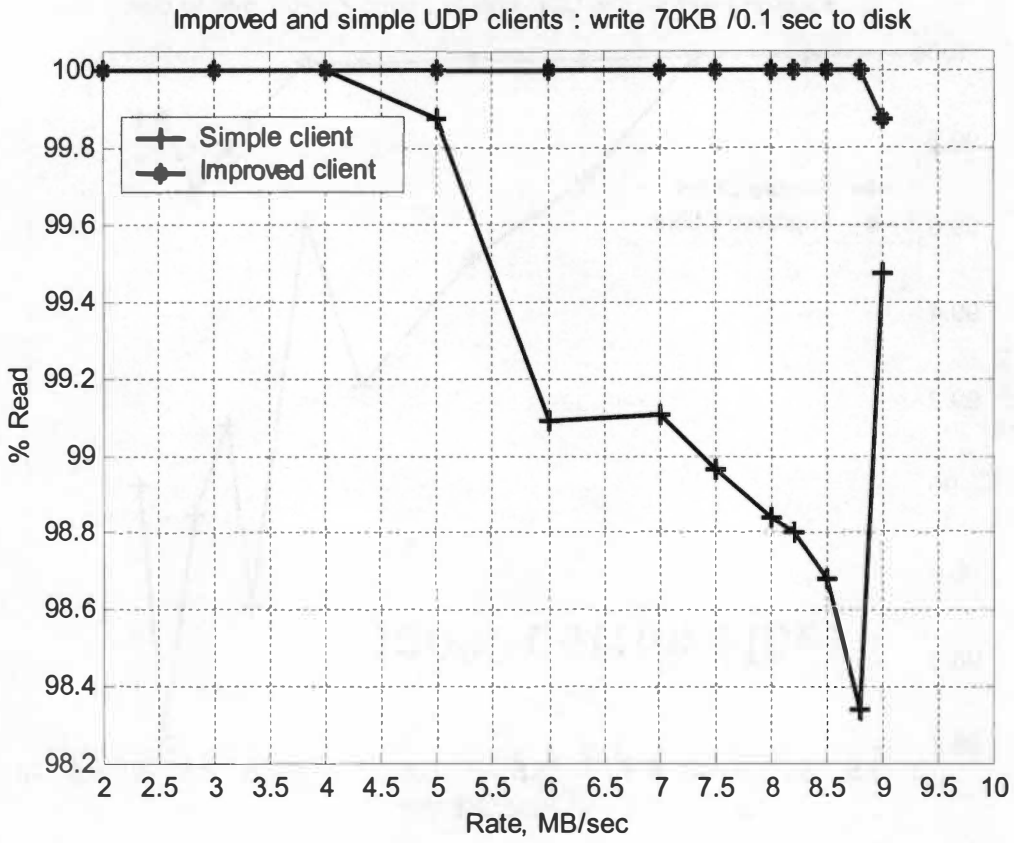

Figure 4.8 Comparing Improved client and simple client when writing to disk with a 50-datagram data buffer.

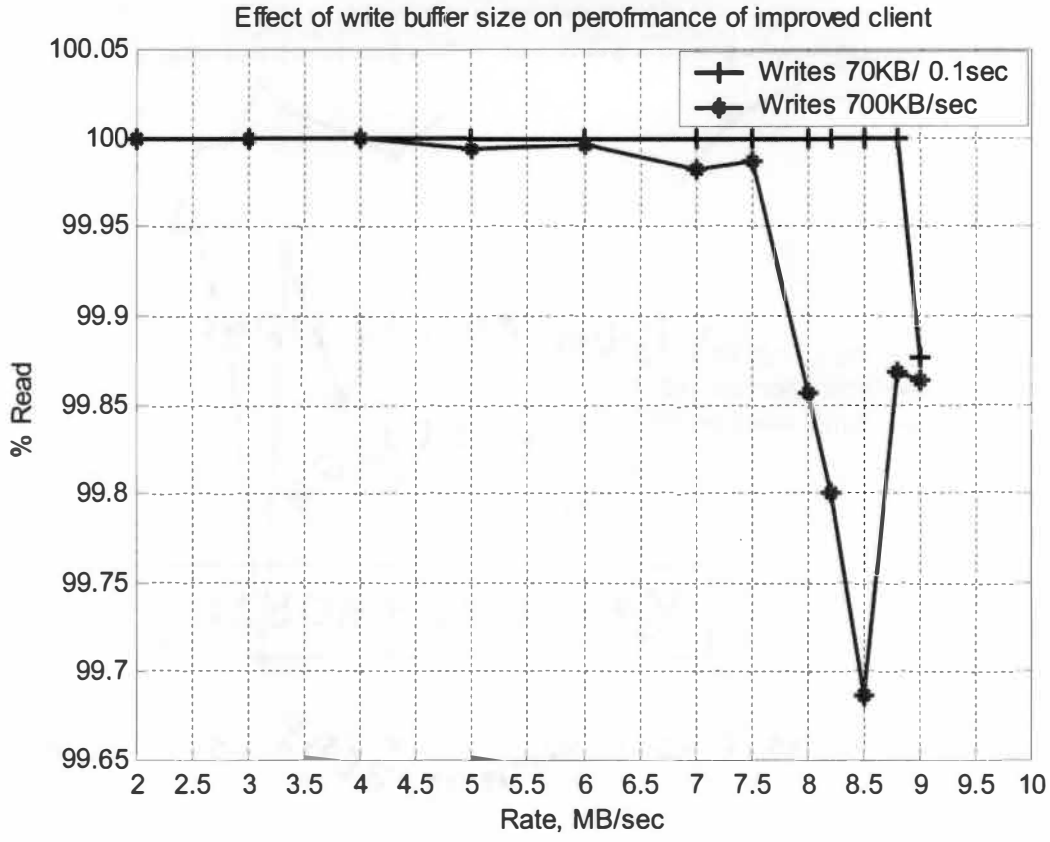

Figure 4.9 Effect of write buffer size on performance of improved UDP client.

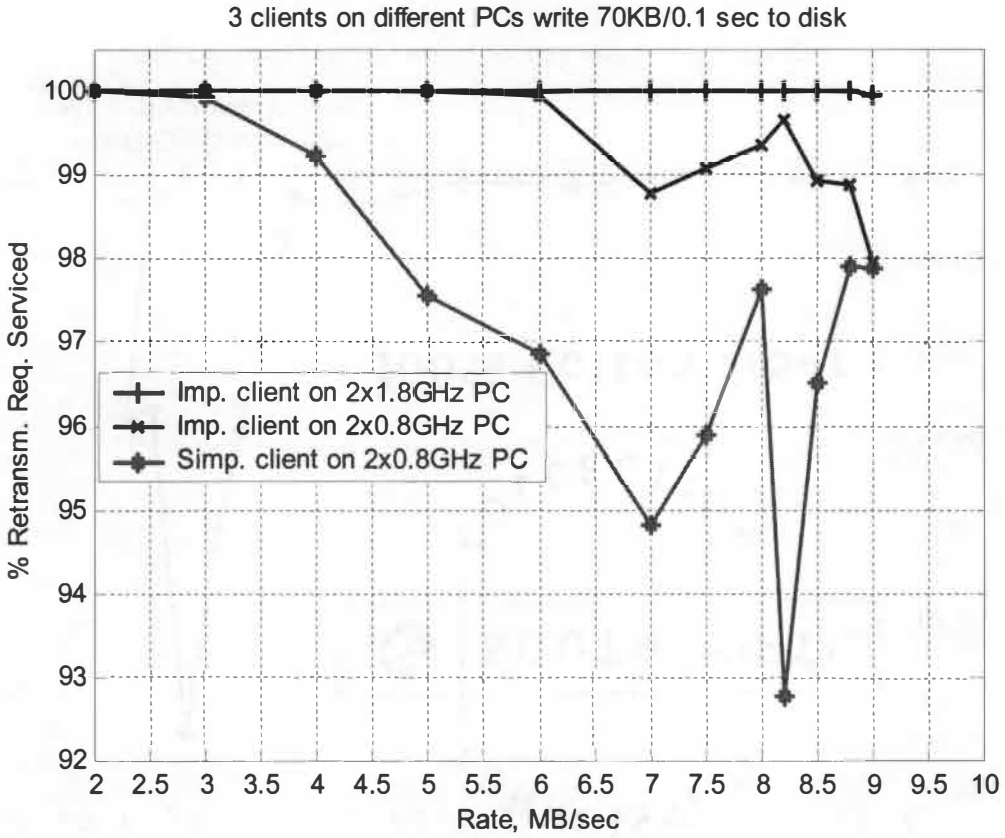

Figure 4.10 Performance of multiple clients being served by an improved server at the same time.

Even though the server was able to respond to the retransmission requests made by the clients for test rates of 9MB/sec and below, figure 4.11 shows that it was able to service 100% of all requests from its three improved clients only at rates of 7MB/sec and below; for 2 clients the maximum rate rose to 7.5MB/sec and for a single client, 9MB/s. As the number of its clients increases and the number of retransmission requests with it, the server gradually becomes unable to respond to all the requests it receives.

4.2 NONBLOCKING UDP SERVER WITH BUFFERS

Another approach that is worthy of note is the case when the server uses a nonblocking mode for the socket that responds directly to client retransmission requests. This then replaces the need for a worker thread because the function call to receive will not block the server application. Even though this is a possible solution, there is a drawback to it. Namely, it cannot buffer client requests because it can only handle one request at a time. The loop must check for the time stamp after each sending effort. If two requests come in between sending operations, the latest will overwrite the older request.

Putting the socket into nonblocking mode and therefore making the socket poll for data is considered a bad programming practice because it uses a lot of CPU time. When the nonblocking server was programmed in C/C++ and tested on the 2x933 MHz Pentium III computer, the CPU usage went up from 19% to over 40%. In view of the above limitations, the nonblocking approach was discarded.

4.3 HANDSHAKING APPROACH

It is possible to code the server application such that it would always wait for an acknowledgement (ACK) from the client for a given number of packets. This would be similar to TCP client-server interaction except that TCP does this very efficiently at the kernel level. For the server to wait for an ACK from the client using UDP socket will

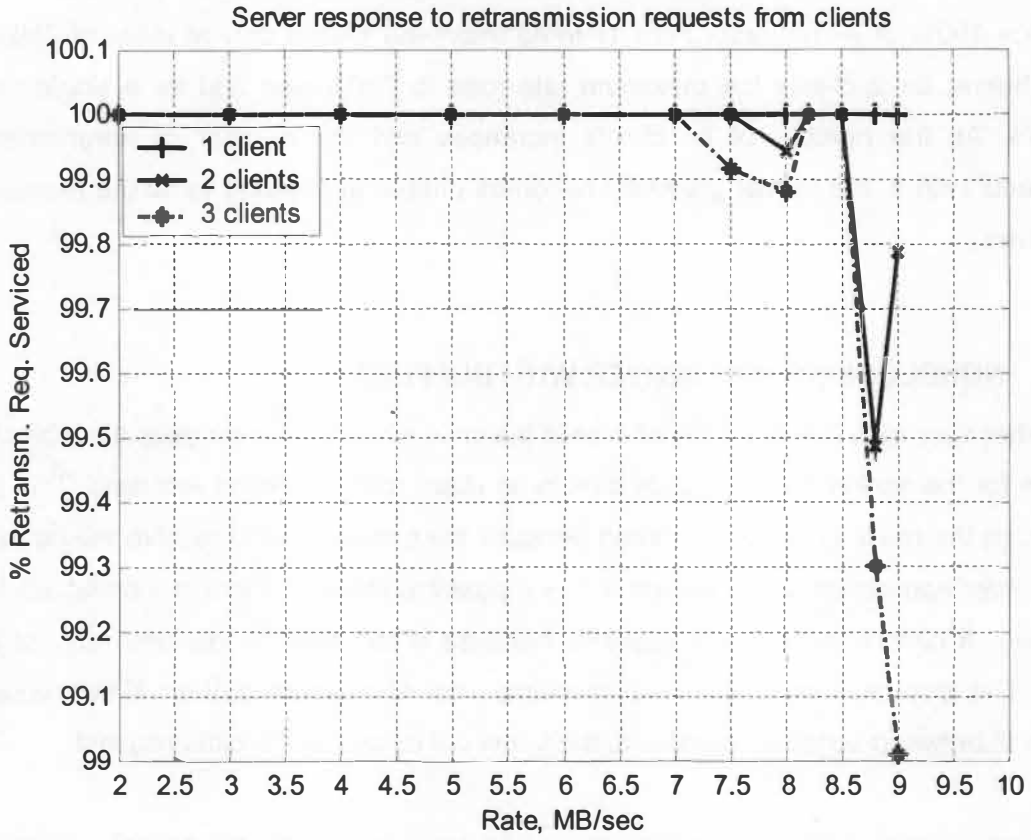

Figure 4.11 Performance of the improved server with 3 clients based on the percentage of retransmission requests it was successfully able to send.

fail since these ACKs may never get to the server just as datagrams sent may never get to their destination. Since a call to receive by default blocks, the server waiting to receive the ACK from its client will block. The server will therefore not be able to function properly with this implementation. As a result this approach was not implemented.

4.4 ANALYSIS OF RESULTS

4.4.1 Interference of improved server due to backlog

Using UDP at rates below 9MB/sec generated no backlog and this situation did not change during the tests shown on figure 4.5 where the improved server was serving three improved clients. This situation could change however if each client PC is engaged in CPU intensive activity such as writing data to disk. From table 4.4 it can be seen that at 8.2 MB/sec the total number of datagrams retransmitted to the three clients amounted to nearly 14000 out of the total of 100,000 sent. This is 14% of the total data sent normally and it is a great concern. Any condition that causes the clients to be losing such an amount of data as to cause the server to be spending 50% of its processor resources on retransmissions should be avoided. The server cannot be effective in such a situation

4.4.2 Ability to handle multiple clients

As noted above, UDP easily solves the question of multiple clients using broadcast. This provides all clients with the same copy of the data at the same time without the need to send individual identical data to all.

4.4.3 Reliability

How successful the UDP client-server implementation is depends on what the client is doing when the data gets to its destination. The results in figure 4.9 show that writing data in smaller blocks gives a better result. This is because writing a larger block makes

it more probable that this disturbance will persist and cause the client to lose the retransmission data when it gets to the client. Since the client only makes one retransmission request, this datagram will be lost forever. So even though writing at a rate of 0.1 sec seems tedious, it still gave better results than writing 10 times slower in larger blocks.

4.4.4 Maximum number of clients supported by the improved server

Under the current method of implementing the improved server, the maximum number of UDP clients in a client-server interaction is approximately determined as being less than B/P, where B represents the size of the data buffer used in servicing retransmissions and P is the average size of the requests made by the clients. This relation is necessary in order to allow for valid retransmissions whenever the need arises. A full development has been given in section 4.1.3.

CONCLUSION

The primary objective of this thesis was to develop techniques for publishing real-time data on a LAN without interrupting the server which is doing time-critical operations. It was desired also that the clients receive most of the published data. To achieve this, application programs were written in C/C++ that implemented TCP and UDP sockets. Standard TCP and UDP sockets were evaluated for publishing and reading data on the LAN. The following conclusions were made :

1) Ordinary TCP sockets blocked the server whenever a client delayed.
2) TCP did not support one-server-to-many-clients connection without reducing the available bandwidth.
3) Ordinary UDP did not guarantee data delivery.
4) Ordinary UDP server never blocked.
5) UDP supported broadcast, which allowed a one-server-to-many-clients data communication.

Using a program written in C/C++, a TCP server with worker thread was developed and evaluated. The worker thread was responsible for maintaining the virtual connection between the server and the clients. This approach was tested and found to prevent the server from being blocked by the client(s) at all times. It also allowed the server to maintain a one-server-to-many-clients without blocking.

For the UDP sockets, software was developed in C/C++ that allowed the improved UDP client to be able to detect and make a single retransmission request for datagrams that it loses. It was determined that the server efficiently serviced retransmission requests from the clients for data rates up to 9MB/s. If the client application was engaged in activities other than reading data from the server, an ordinary client read 100% of all published data only at rates below 4MB/s, while the improved client read 100% of data up to 6MB/s

A future study may be directed toward resolving questions about the number of retransmissions that a client must make. That is, will a second retransmission effort by an improved UDP client be able to raise the maximum rate at which it reads 100% of the published data ?

LIST OF REFERENCES

LIST OF REFERENCES

1] Beveridge, J. and R. Wiener. *Multithreading Applications in Win32: The Complete Guide to Threads.* Addison-Wesley Developers Press, 1998.

2] Comer, Douglas E. *Computer Networks And Internets With Applications,* 3rd ed., Prentice Hall Upper Saddle River, NJ, 2001.

3] Jones, Anthony and Jim Ohlund. *Network Programming for Microsoft Windows*. Microsoft Press, 1999.

4] Lannon, John M. *Technical Communication*, 9th ed. Longman, NY, 2003.

5] "Microsoft CE.NET Socket Functions" Aug. 2002, *Microsoft Library*, Jan-May 2003, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcewinsk/htm/ ceconsocketfunctions.asp>

6] "Microsoft CE.NET Winsock Structures" Aug. 2002, *Microsoft Library*, Jan-May 2003, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcewinsk/htm/ceconmicrosoftwindows-specificextensionfunctions.asp

7] "Protocol Directory: TCP/IP Suite" Feb. 2003 <http://www.protocols.com/pbook/tcpip.htm#TCP>

8] Quinn, Bob and Dave Shute. *Windows Sockets Network Programming*. Alan Feuer. Addison-Wesley, 2002.

9] Schildt, Herbert. *C++: The Complete Reference*, 3rd ed., Osborne McGraw-Hill, CA, 1998.

10] Snader, Jon C. *Effective TCP/IP Programming : 44 Tips To Improve Your Network Programs*. Addisson-Wesley Developers Press, 2000.

11] Stroustrop, Bjarne. *The C++ Programming Language*, 3rd ed. Addison-Wesley, 2001.

VITA

Victor Anderson was born in Kumasi, Ghana on June 19, 1967. He spent most of his early childhood years in Tema, a city close to the capital city Accra. He had his elementary education in Tema Community 8 Primary and Middle School from 1973-1981. He attended Suhum Secondary Technical School and St. Augustine's College in Cape Coast from 1981-1988. Gaining a scholarship from his government to further his studies, he went to Russia. In 1996 he graduated from Moscow Power Engineering Institute with an M.S. in electrical engineering.

From 1996 to 2001 he worked as an electrical engineer at an electrical utility company in Ghana. Developing an interest in electronic engineering, he enrolled at the University of Tennessee Space Institute in 2001, where he completed the MS degree in electrical engineering in August 2003.